IQM4HD Concepts

Thomas Oelsner, Felix Heine, Carsten Kleiner

August 23, 2019

Contents

1. Introduction 5
1.1. Data Quality 5
1.2. A DSL for DQ monitoring 7
1.3. Overview 9

2. DSL 10
2.1. Requirements 10
2.2. Concepts 11
2.2.1. Basic Design Decisions 11
2.2.2. Data Model 12
2.2.3. Sources 13
2.2.4. Checks 19
2.2.5. Actions 23
2.2.6. Roles 24
2.2.7. Return values 28
2.3. DSL Description 30
2.3.1. Expressions 30
2.3.2. List expressions 33
2.3.3. Built-in Methods 36
2.3.4. Sources 37
2.3.5. Checks 39
2.3.6. Actions 42
2.4. Environment 44
2.5. Optimization 44
2.6. Evaluation 45

3. Complex DQ Rules 48
3.1. Parametrization 49
3.2. Univariate Time Series 49
3.2.1. Implementation 50
3.2.2. Seasonal time series 56
3.3. Multidimensional Data 59
3.3.1. DSL extensions 60
3.3.2. Profiling 62
3.3.3. Running the check 62

4. Profiling 64
4.1. Introduction 64

4.2. Profiling, Statistics and Rule Generation . . . 65
4.3. Profiling Methods . . . 66
4.3.1. NOT NULL profiling . . . 66
4.3.2. Range profiling . . . 67
4.3.3. Foreign key dependencies . . . 67
4.3.4. Time series profiling . . . 67
4.3.5. Cube profiling . . . 68
4.4. Design . . . 68
4.5. Architecure . . . 69
4.5.1. ProfilingMethod . . . 69
4.5.2. ProfilingResult . . . 71
4.5.3. API . . . 71

5. Feedback Loop **73**
5.1. Return values from checks . . . 73
5.2. Rule types . . . 74
5.3. Re-profiling for individual actions . . . 74
5.3.1. Re-profling for NOT NULL . . . 74
5.3.2. Re-profiling for RangeCheck . . . 75
5.3.3. Re-profiling for percentage NOT NULL . . . 75
5.3.4. Re-profiling for time series . . . 75
5.3.5. Re-profiling for cubes . . . 76

6. Related Work **77**
6.1. Existing solutions for data quality monitoring . . . 77
6.2. Individual implementation aspects . . . 78
6.3. Advanced rules and profiling . . . 79
6.3.1. Alte version . . . 79

7. Conclusion **81**
7.1. Open Issues . . . 81

Appendix **82**

A. Rulecatalog **83**
A.1. Column property checks . . . 83
A.1.1. NotNullCheck . . . 83
A.1.2. RangeCheck . . . 84
A.1.3. BoundaryChecks . . . 85
A.1.4. PatternCheck . . . 85
A.1.5. ContainsCheck . . . 87
A.1.6. NestedDocumentValueCheck . . . 87
A.2. Structure analysis checks . . . 88
A.2.1. QuantityCheck . . . 88
A.2.2. SumCheck . . . 88

A.2.3. UniqueCheck 88
A.2.4. Referential Integrity 89
A.2.5. Functional Dependency 89
A.3. Value checks 90
A.3.1. DistributionCheck 90
A.3.2. ARCheck 91
A.3.3. CubeCheck 92

1. Introduction

Both data as well as information derived from it has an increasing impact on companies, since many short- and longterm decisions are based on the interpretation of data. In addition to the long established relational databases, unstructured or semi-structured data sources are gaining importance. Examples of such unstructured data are Weblogs, data from social networks or RFID-data. Therefore, the relevance of data quality monitoring is increasing enormously for these kind of data sources, in order to prevent wrong decisions based on faulty analysis due to flawed data.

Main target of the *IQM4HD*[1] joint project is the study of advanced concepts for data quality monitoring of heterogeneous data sources. The concepts are implemented in a prototypical tool for data quality analysis. At the heart of the solution, we define a DSL called *RADAR* that is used to describe data quality rules. This tool consists of an interpreter for this DSL, components for semi-automated detection of data quality rules, as well as the automated monitoring of the defined quality rules.

Furthermore, focus is laid on the area of detection and derivation of data quality rules. One goal is to automatically detect complex statistical dependencies and suggest rules based on the results. The suggested rules will be presented using the *RADAR* language. The data quality manager supervises these automatically created rules and may adjust them. This is introduced in order to improve the rule base continuously.

A central data quality repository is introduced, which contains data quality rules as well as metadata on rule checking executions. Besides the semi-automatically detected rules, the data quality manager may insert manually created data quality rules. The permanent data quality monitoring uses the rules from the central data quality repository and regularly checks those against current database content. Deviations will be logged and reported.

In the following section of this introduction, we explain our understanding of data quality. Then, we motivate our approach of using a special DSL for quality monitoring. Finally, we give an overview of the rest of the report.

1.1. Data Quality

In order to provide solutions for data quality monitoring, one first has to define data quality. The general idea of data quality is, that if data fits the requirements for its intended use, it has quality. Complementary are several different dimensions of DQ to describe these requirements. It has been a broadly researched topic [30, 29, 25] and presented in [25] are sixteen different DQ dimensions including *Security, Ease of*

[1] http://iqm4hd.wp.hs-hannover.de/english.html

Manipulation, Objectivity or *Free-of-Error* among others. In this project, we focus on specific aspects of data quality, as we target quality *monitoring*. With our solution we want to monitor DQ with regard to the data's content, meaning the actual values and their relationships. This DSL is primarily designed for the dimensions *Completeness*, *Consistent Representation* and *Free-of-Error*, due to our goal to cover faulty data. On the contrary, Dimensions like *Security* or *Objectivity*, besides others, are not of interest to us, since they concern a completely different domain respectively.

Another point of view to describe our work is that we focus on complex constraints that are defined with our DSL. Normally, database constraints are unable to express all kinds of complex integrity constraints defined on a conceptual level. This is mainly due to two reasons: On the one hand, database constraints have limited expressivity, and on the other hand, they are evaluated during online operation and thus have negative performance impacts. The latter issue often leads to disabled constraints due to performance bottlenecks. Here, our approach comes into play to provide a monitoring system that checks constraints offline in regular intervals that are either not expressible in that database constraint language or that cannot be activated due to performance considerations.

Furthermore, our system can check constraints that span multiple databases, even from different vendors and with different query languages. Thus, we also allow to test the contents of a database against some reference database, or to test the mutual consistency of different databases.

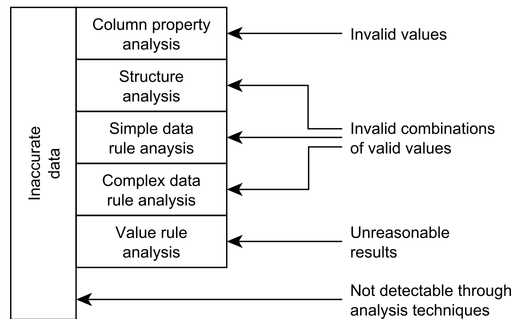

Figure 1.1.: Different types of inaccurate data as described by Jack E. Olsen [24].

Our approach uses data quality rules to create a foundation with which to monitor data continuously. These data quality rules are supposed to cover the aspect of data quality mentioned beforehand, i.e. invalid values. Therefore, we base them on the description of inaccurate data by Jack E. Olson [24]. He divided inaccurate data in five different categories which are shown in figure 1.1 and are described in the following:

- *Column property analysis* considers single column values and is about the validity of a single data point. This contains rules like null- or range-checks.
- *Structure analysis* includes, among others, investigations of primary/foreign key

pairs and redundant or synonymic data columns. In essence it handles structural constraints spanning multiple columns. It gives information about a set of data rows violating a rule, instead of which particular data point does.

- *Simple data rule analysis* reflects the analysis of valid values with regard to a business object. It goes beyond structural validation to semantic validation.
- *Complex data rule analysis* is analogous to simple data rule analysis with the addition of checks spanning multiple business objects in contrast to just single business objects of the aforementioned method.
- *Value rule analysis* is about non binary results. It checks whether the data set as a whole could be valid including operations like checks for cardinality, counts, averages, medians and so on. The result is a non binary answer, it leaves room for interpretation.

Even though Olson uses these categories in a context of data profiling in order to identify inaccurate data, we use them as a boundary to specify our area of data quality monitoring. This is partly owed to the fact that a major component of our framework is concerned with the analysis of data in order to propose rule candidates.

1.2. A DSL for DQ monitoring

In order to detect quality problems reliably, it is important to check the data regularly. An extreme solution would be to run checks as database constraints, effectively avoiding the insertion of wrong data. However, this approach has multiple drawbacks: 1) The check logic slows down all processes that modify the data. 2) The check logic must be implemented inside the database and thus depends on the capabilities of the database. Not every check can be implemented on every system, especially more complex checks like statistical checks. 3) These kinds of checks often rely on proprietary languages like stored procedure languages. 4) Implementing these checks in a modular way, in order to apply the same logic to different databases, is often impossible. 5) Some checks, esp. statistical checks, do not always indicate clear errors. They rather result in warnings indicating an unusual data distribution that should be checked. Thus always rejecting non-conformant data is not possible. 6) Checks that span multiple databases (e.g. a relational database and a document database) are impossible with this approach.

Thus, our approach is to run the checks from an external engine that has its own language to encode the checks, called **rules** in our system. For this, we designed the *RADAR* DSL. In order to motivate our approach, we start with an example of a more complex DQ rule. Assume you a customer base and you want to check whether the date of birth is correct. Just checking ranges (i.e. no customer must be older than 100 years) would miss important quality problems. Assume e.g. you import customer data from another source where an unknown date is encoded wrongly as 1970/01/01. This is still a valid birth date, so that this error would never be detected using a simple

Class	Age range	Probability
$j = 1$	$[0, 18)$	$p_{01} = .05$
$j = 2$	$[18, 25)$	$p_{02} = .20$
$j = 3$	$[25, 35)$	$p_{03} = .15$
$j = 4$	$[35, 50)$	$p_{04} = .25$
$j = 5$	$[50, 65)$	$p_{05} = .19$
$j = 6$	$[65, 80)$	$p_{06} = .10$
$j = 7$	$[80, 95)$	$p_{07} = .05$
$j = 8$	$[96, \infty)$	$p_{08} = .01$

Table 1.1.: Reference age distribution

range check. For this, a distribution check that assesses whether the distribution of customer's ages follows a pre-known, typical distribution would be able to detect a larger number of customers with birthdate 1970/01/01.

The idea behind this rule is that we have a known age distribution for our customers, which has been derived from historic data. We now want to check continuously whether the customer base still matches this distribution. The mathematics behind this check is a $\mathcal{X}^2$ test. First, we need the reference distribution. It is defined as probabilities for the age classes, see table 1.1. We have $m = 8$ classes with a probability p_{0j} for each class.

Now we have the current customer data, consisting of n customers with age information. We count the number of customers in each class N_j. We then compute the expected number of customers in each class as $n_{0j} = p_{0j}n$. The test statistic is computed as follows:

$$\mathcal{X}^2 = \sum_{j=1}^{m} \frac{(N_j - n_{0j})^2}{n_{0j}} \tag{1.1}$$

Under the null hypothesis (the customer age is distributed according to the reference distribution), this value follows a chi-square distribution with $m - 1 = 7$ degrees of freedom. This means, that we have a 95% chance that $\mathcal{X}^2 < 14.07$ under the null hypothesis. Thus 14.07 might be a good threshold to issue a data quality alert.

Now assume the data is stored in a relational table `customer(id, name, dob)`, where dob is the date of birth, and the reference distribution is stored in another table `ref_age_dist(j, from, to, pj)`. Using SQL, a check can be written as follows:

Listing 1.1: SQL check for age distribution.

```
SELECT * FROM (
  SELECT SUM( (Nj-n*pj)*(Nj-n*pj) / (n*pj) ) Xsq
  FROM (
    SELECT d.pj, COUNT(*) Nj,
           (SELECT COUNT(*) FROM customer
            WHERE dob IS NOT NULL) n
    FROM (SELECT TRUNC(months_between(sysdate, dob)/12)
```

```
            AS age
            FROM customer WHERE dob IS NOT NULL) c
      JOIN ref_age_dist d
            ON c.age >= d.from_age
            AND (c.age < d.to_age OR d.to_age IS NULL)
      GROUP BY d.j, d.pj
   )
) WHERE Xsq >= 14.07;
```

However, this code combines several aspects in one single statement. The statement contains code to access the data (`customer`), access the reference distribution (`ref_age_dist`), the preprocessing of the customer data (filter customers without dob, convert dob to age), the aggregation of customer data to age classes, the logic to compute the test statistic, and the final evaluation of the threshold. This has several drawbacks:

- The code is hard to understand due to missing modularization. The different user types explained in the previous section are all merged in this single statement and thus will have to create it together without any separation of concerns.
- The logic is not reusable. In order to check another data source against another distribution, the whole code has to be copied and adapted.
- The code is specific for relational databases. The same data e.g. in a MongoDB system cannot be checked. In this case, it even contains system-specific code (Oracle functions months_between, sysdate), thus just porting it to PostgreSQL would mean to modify the code.
- The code must be executed on the target database.

Our approach and the DSL *RADAR*aim to remedy these drawbacks.

1.3. Overview

The remaining report is structured as follows. ...

2. DSL

This chapter introduces the proposed language, that we call *RADAR*. The goal of the DSL is to create a basis with which to easily compose data quality rules for identifying the inaccurate data described in the previous section. First, we explain the requirements for the DSL that drive our design decisions. Then we explain the basic design and describe the main concepts of the language. In the subsequent section, we specify the syntax and semantics of the language in depth.

Furthermore, we describe the rule catalog that contains many pre-defined types of quality checks that can modularly be used in projects, like a standard library in other languages. In the section "Environment" we describe the interaction with the environment, e.g. the scheduler that initiates the invocation of the rules. The final section describes implementation details, including the parsing process, the internal representation, the execution engine, and optimization techniques.

At first we look at what information is needed to compose data quality rules, and secondly which roles are present in our system. Information needed for quality rules includes (i) *what data should be checked*, (ii) *what quality aspect should the data be checked for* and lastly (iii) *what should be done with the result.* Regarding roles, we suggest a separation between the technical aspects of data retrieval and the subject-specific knowledge needed for writing coherent rules. Firstly, the technical user with knowledge about database relevant issues and secondly, the data quality manager with knowledge about the data's meaning. Whereas the technical user just prepares the data for the data quality manager, the latter will - with his knowledge about the data - compose data quality rules or adjust automatically generated rules.

For a start we outline the requirements a language for data quality rules should fulfill and consecutively we introduce each aspect of the language itself, concluding with a summary of the results.

2.1. Requirements

In order to design a rule language that fits our ideas we start with defining requirements the language has to fulfill:

1. **Reusability** First, the constraint logic has to be defined in a flexible and reusable way. This means, we need a language that supports the definition of various constraint types in a concise manner. It further means, that the logic behind the constraint is decoupled from the actual database and DBMS to which the constraint is applied.

2. **Optimizability** For efficient execution, it must be possible to execute most of the logic of the constraints directly on the database using DBMS features. However, in case of limited resources of the database server, or in case of limited capabilities of the DBMS, it must also be possible to execute the logic within our system. So the language cannot specify the exact way of executing a constraint check and has to leave room for optimization by the execution logic.

3. **User groups** In our system multiple persons with different expertise will be involved. First, there are experts for the data sources to be monitored, that understand the data models of these systems. Second, we have domain experts that understand the business logic and can define which data is valid from a business point of view. Third, there are programmers that understand how to code the business logic programmatically. We need the respective parts of the language to be accessible by the corresponding groups.

4. **Heterogeneous Data Sources** The data quality rules should be independent of the underlying data source where the actual data is stored. For instance, the same rules should be usable for customer data originating from a relational database system as well as originating from a document database. Thus, the rules can neither be specified in the database's own language nor rely on the data being accessible in a specific data model at all.

5. **Advanced Quality Rules** Apart from rather simple data quality checks such as non-null or range checks the rule language should also provide the option to specify advanced data quality checks. Examples of these types of checks that we implemented in our prototype include cube analysis checks that determine dimensions in a data cube providing explanations for data quality problems that reflect only on a higher aggregation level in first place. Another example is a time series check where outliers in time series according to a flexible series model are discovered.

6. **Extensibility** Not all desired quality checks are known at design time. For example the advanced quality rules explained above have only been discovered throughout the course of the project. Consequently, the *RADAR*language has to be extensible. That means apart from a fixed set of provided data quality rules, developers have to be able to implement their own data quality rules with the language in an easy way, which can then be executed in the same way as the provided rules.

2.2. Concepts

2.2.1. Basic Design Decisions

Derived from aforementioned requirements combined with the knowledge about what information is needed for data quality rules, our proposed DSL will be split into three main parts; **Sources**, **Checks** and **Actions**. Sources are meant for data description,

whereas Checks depict archetypes of data quality rules, like a simple check for null values. The Actions combine Sources and Checks to form data quality rules. Whereas Sources and Checks are more on the technical side, Actions were designed with natural language in mind, so that the purpose of any given Action is obvious. The following subsections will introduce each aspect of the DSL and give a few examples on how they may look.

The Source layer will be defined by the source system experts and defines a uniform access layer providing those schema objects for the next layers that are subject to quality monitoring. Within the sources, queries in the native language specify how data is retrieved from the system. Each Check basically defines a logic that specifies whether given data contains quality problems. The Check is decoupled from concrete sources and is defined using a flexible parametrization. Actions, finally, define which Checks will be called using which data sources (and probably other parameters specifying the behavior of the check). This decoupling solves the first requirement, as Checks are reusable for different actions. Furthermore, the third requirement is solved as programmers can define Check logic, while the domain experts define actions. The second requirement, optimizability, is solved by declarative semantics in the Check language together with the implementation of the execution engine. The details of this implementation will be described later.

There will be two kinds of Checks. First, we have general purpose Checks that define logic that is useful in many databases. These Checks include standard column checks like range or NULL checks, structural checks like foreign key checks, participation constraints and uniqueness tests. Also standardized statistical tests like time series tests or distribution tests fall into this category. We aim to provide a growing set of standard checks that can be used out of the box. We call this set the rule catalog. Second, we will have domain specific checks that describe specific business rules for some application domain. These checks are typically written by programmers during a project. The domain experts then can use this toolbox of various checks to define quality rules on the data.

An important validation for the language is to show the reusability of checks. This is done by showing that the parametrization of Checks is flexible enough so that all predefined Checks can be used on two completely different target databases without the need to write new check logic, only by applying existing Checks to the defined Sources.

With the modularity comes another advantage: it allows for expandability in two ways. Firstly, it allows us to progressively add support for different kind of data, starting with relational data, ranging to NoSQL DBMS or even plain documents. Secondly, it allows for extensibility with regard to rule types.

2.2.2. Data Model

The basic data type of data in our DSL is a **record**. These records are similar to the relational model, however, with one important distinction. Each record carries so-called *roles* instead of attribute names. Each attribute can have multiple roles, and each role can be assigned to multiple attributes. As an example, the following

illustrates a customer record:

group identifier	id identifier	firstname name	lastname name	birthdate	telephone
A	12345	Jon	Doe	5/7/1980	111-222-33333

In this case, the first attribute has the roles "group" and "identifier". However, the role "identifier" is assigned to the first two columns. When we access this role in our DSL, we effectively access the two-attribute record ('A', 12345). This is a very flexible and convenient way to write code that can handle either attributes or compound data using a single name / parameter. The order of the attributes is important. The attributes of the record have basic data types, like strings, integers, doubles or booleans, and there is a NULL value. There is no explicit data type definition, each value carries its own type information. A non-existing role in a record is implicitly treated as value NULL.

The next higher level structure is a **list** of records. The records in a single list do not need to be homogeneous, each record can have a different set of roles and the data types of the same role could vary across the records.

This way, we can map both structures from relational databases as well as document-oriented databases to our internal model. Even other data sources are supported as long as they can be mapped to a list of records. The relational mapping is straightforward, while the mapping from documents flattens the documents. Sub-documents are mapped into a flat structure using role names with dot notation. Arrays are either ignored or an unwind operation (see e. g. `https://docs.mongodb.com/manual/reference/operator/aggregation/unwind`) is used to map each array element to a new instance of the surrounding document together with this element, according to the needs of the quality check logic.

2.2.3. Sources

Sources are the interface to databases or other reference values a domain expert may work with when composing data quality rules. There are two main types of Sources. On the one hand there is the domain mapping layer, which maps database structures to Sources, like the customer Source from listing 2.3, referring to a customer table. On the other hand, there are constant values, patterns or complex parameter sets that e.g. describe statistical models. These constants are also defined as Sources for use in data quality rules. An example for this kind of Source is shown in listing 2.1, an email pattern. This pattern could be used to check the customers emails for invalid entries. A statistical parameter set could be a histogram containing a reference distribution to check the age distribution of customers, as shown in listing 2.2. The unification of patterns, constant values and the domain mapping layer to Sources was introduced to simplify the work of domain experts.

Listing 2.1: DSL code for an email pattern.

```
SOURCE EmailPattern TYPE CONST ROLES (email):
```

```
  "[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}"
END
```

Listing 2.2: DSL code for a reference historgram.

```
SOURCE AgeHistogram TYPE LIST CONST ROLES (from, to, bin,
    ↪ perc):
 [0, 18, "0-18", .05],
[18, 25, "18-25", .20],
[25, 35, "25-35", .15],
[35, 50, "35-50", .25],
[50, 65, "50-65", .19],
[65, 80, "65-80", .10],
[80, 95, "80-95", .05],
[95, NULL, "95-", .01]
END
```

The domain mapping layer is the focal point of Sources. It was designed to create an interface between the technical and the domain-specific side of data quality rules. The goal is to provide an easy way for domain experts to work with the underlying data. Since Sources are written in the databases own language, it is an assignment for technical users to prepare Sources for domain experts. The customer example in listing 2.3 shows a generic SQL statement selecting the id, name, dob (date of birth) and email amongst others from the customer table. Additional meta information is added by declaring the field *id* as the identifying value of this data construct. The identifier `cust` references a database connection that has to be configured in the system.

Having the source layer as an intermediate layer between rules and the target database allows a very flexible way to define data access for any system using the system owns query language and to map structures to the domain layer accessed in the next parts of the DSL.

Listing 2.3: DSL code for a Customer Source.

```
SOURCE Customer TYPE LIST QUERY ROLES (id: IDENTIFIER):
DATABASE cust NATIVE
  SELECT id, name, firstname, dob, email,
                 title, bonus
  FROM customer
END
```

Beside the simple selection of e.g. tables and fields, it is also possible to do a pre-filtering of data based on technical structures. An example could be a large unified table containing data from different branches in various countries. The table contains a field that divides the data by country. A Source definition could look like the customer example in listing 2.3 with the addition of "`WHERE country LIKE 'DE'`" at the end of the SQL statement. Being able to restrict Sources this way has two advantages, the

first one is the avoidance of legal issues, whereas the second benefit considers semantic issues. A domain expert working for e.g. the German branch of the company may only be legally allowed to view and inspect customers of German branches. The restriction of the Source to contain only data tuples with the country code for Germany ensures that. The other benefit, as mentioned beforehand, would be the semantic differences between various data sets. Coming back to the example with a Source only containing German customers, there are semantic differences in the way ZIP codes, telephone numbers or street names (among others), are formatted in different languages. This separation allows for data quality rules, explicitly designed for, in this case, the German instances of the aforementioned data fields.

Sources referring to databases are meant to be a way to describe the structure of the underlying data. The code contained in a Source may be altered pre-execution in order to improve the performance of the monitoring. To provide a performant solution, optimization - ranging from simply executing checks on databases with its performant operations to combining multiple checks in one query - is needed when executing data quality rules.

Resulting from all the aforementioned design choices, a key part of defining domain mapping layer objects can be derived.

> A Source should be defined as simple as possible and should not contain calculations related to data quality checks in its definition.

Additional structural information in form of roles, may be given in the header of a Source, like seen at the `IDENTIFIER` tag in the customer example. To further expand on the `IDENTIFIER` example, it is useful when evaluating the given data with data quality rules. A detected data quality issue may be displayed by showing the violated rule and the tuples leading to the violation. When using more complex analysis methods, Sources may need to be enhanced as well. For example, in time series analysis the frequency of data points has to be known, or an offset may be declared. In data cube analysis, the cube model and other metadata is necessary as well. Constant Sources may use roles as a way to name the constant values, as seen in listing 2.2, the three columns of the record are name “from”, “to” and “perc” respectively.

Mapping of database query results to source data

In the case of using a query for sources the query to retrieve data from the underlying database is expressed in the database’s own query language. Nevertheless the result of such a query has to be mapped to internal structures in order to be used in the defined source in further processing steps of the DSL. The mapping from database query result to source structure depends on the type of external database system used. In this project we use traditional relational databases as well as document database systems as examples. In case the DSL will be extended to other types of external data sources the following mapping considerations will have to be extended to these database types.

Mapping from relational database systems

In the case of relational database systems a query will be a SQL query which in general returns a result relation, i.e. a list of tuple values $(v_1, ..., v_n)$ which are homogeneous. This means that all tuples have identical length n, identical attribute names $a_1, ..., a_n$ for their attributes and each of these attributes a_i has the same datatype τ_i for all tuples. All datatypes τ_i are typically simple datatypes in relational database systems such as numbers, strings, dates, etc. Object-relational features of a database such as array or structured types are not considered here.

Usually the a database query will result in a `LIST` source. So every tuple of the query result will be mapped to a `LIST` element in the source. The a_i are used as the names of the tuples' attributes within the data source by default. They can be renamed by using role names as described above.

If a database query is not defined as a `LIST` source it may only return a single row. This row may internally have the same structure as rows which are part of a `LIST` source. In case the row consists of multiple attributes the result is treated as a record value where each attribute name a_i is used as name for value part v_i of the record. Again, attribute names can be renamed using roles as before. If only a single row with a single attribute is returned by a query the result can be mapped to a `CONST` value which, by default, is named as the attribute's name.

In line with the general approach of our DSL concept there is no declaration of types for the query results and their components. Nevertheless, certain functionality in the later processing steps of the DSL may require data to be of a certain type in order to be usable. Responsibility to ensure that these requirements are met stay at the side of the DSL user. Failure to meet these requirements may result in errors by components processing data in quality checks at runtime.

Mapping from document database systems

Similar to relational database systems, queries in the database's own query language are also used to retrieve data from document databases. In general the result of a query in a document database is a list of documents matching the query. Thus, usually the corresponding `SOURCE` in the DSL will be declared as a `LIST` type where each of the result documents from the query is mapped to a list element in the `SOURCE`.

Listing 2.4: DSL code for a Customer Source from a Document DB.

```
SOURCE CustomerMongo1 TYPE LIST QUERY
                      ROLES (_id: IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.find()
END
```

Nevertheless, there are significant differences between these document objects and a tuple retrieved from a relational database system as discussed above:

1. Each of the documents in the query result may have a different structure, i. e. even though for every single document its internal structure consists of a set of keys

$k_1, ..., k_n$ identifying associated values $v_1, ..., v_n$, both the names k_i of the keys may be different between documents in a single query result as well as the number n of values at all.

2. The datatypes τ_i of the values v_i inside the documents may not only consist of simple types such as strings, numbers or dates, but may also be arrays of values, arrays of documents or a single hierarchically nested structured document themselves.

Navigation into the document structure, i. e. accessing certain values v_i within a document, is done by specifying the corresponding key name k_i which is also assigned as the default role. Regarding the first difference as mentioned above we consider each document individually with its own structure. That means that all documents that do not contain a value v_i for key k_i which is accessed are not considered at all for a certain check as if they were not present in the data set unless the check is a `NOT NULL` check. We assume that data that is not present in a certain document will never be relevant for any data quality check apart from checking for its presence. Thus, for `NOT NULL` checks, missing keys are assumed to have a `NULL` value. For all other checks documents missing a certain key k_i are not considered.

For potentially large documents it is advisable to select only those keys from the documents that will be used within this source in the other parts of the DSL. This is, similarly to the projection in the `SELECT` part of a SQL statement by using a projection operation inside the database-system specific query. An example for such a projection in Mongo DB is shown in listing 2.5.

Listing 2.5: DSL code for a Customer Source from a Document DB with Projection.

```
SOURCE CustomerMongo2 TYPE LIST QUERY
                      ROLES (_id: IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.aggregate( [ {
    $project : {
      "name" : 1, "dob" : 1,
      "email" : 1, "ccn" : 1,
      "default_payment" : 1, "ssn" : 1
    }
  } ] )
END
```

Note that the heterogeneous structure of the documents does not impose any ordering on the keys, i. e. it does not make a difference whether a certain key k_i occurs at the same position within the document or not. Values are solely identified by a string matching of their key names (or assigned role names, if any).

Note in addition that even though documents can be heterogeneous in structure, often documemt database systems assume the existence of a specific key in every document whose value can be used to uniquely identify a document (such as the key

_id in MongoDB). This key is often a good choice for the IDENTIFIER attribute of a source.

For documents containing values which are of the array type as described above the DSL does not provide extensive handling capabilities for the arrays. One reason is that typically many of the relevant array operations such as unnesting of arrays or accessing specific elements inside an array are already provided in the document database query language. Thus these operations can easier and more efficient be included into the query defining a data source than re-implementing them based on the Source in our DSL. For example in listing 2.6 we specify a Source where customer documents are assumed to have a key address which is an array of nested documents containing key-value pairs for each address of a customer. Due to the smart implementation of unwind on the side of the document DB, single addresses do not even have to be stored inside an array, but can just be a nested document. This is because unwind handles single values in the same way as an array containing a single entry.

Listing 2.6: Customer Source from a Document DB flattening an array of addresses.

```
SOURCE CustomerAddress TYPE LIST QUERY ROLES (_id:
   ↪ IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.aggregate([ { $unwind: "$address" } ])
END
```

This query implicitly creates a document with the exact same key-value pairs and one of the elements of the address array in addition. This behaviour makes it impossible to use the document id _id as identifying information for this source as multiple elements of the Source list will have the same value. If an identifier is required (which will often be the case), query functionality of the underlying document database should be used to create a composite identifier consisting of document _id and the position inside the array value. Listing 2.7 shows how this is specified in Mongo DB. The position of each address inside the array is added to the returned document as key adrPos, thus creating a unique key for the document in the combination of _id and adrPos.

It should be noted though that, depending on the size of the documents and the length of the array, this technique may significantly increase the overall size of data to be generated in a Source. The worst case size of such a source is proportional to the sum of the product of the largest document size and the length of the longest array. Thus, as explained above a reduction of the document size by using an appropriate projection should be considered wherever applicable. In addition, defining multiple Sources for different aspects of a document collection may be more efficient in this case when compared to defining a single source for the whole document collection.

Listing 2.7: Customer Source from a Document DB flattening address array with Identifier.

```
SOURCE CustAddWithPos TYPE LIST QUERY
                      ROLES (_id: IDENTIFIER,
                             adrPos: IDENTIFIER):
```

```
DATABASE custmongo NATIVE
  db.customer.aggregate([
    { $unwind: { path : "$address",
                 includeArrayIndex : "adrPos" } } ] )
END
```

Finally, the behaviour regarding returning documents that do not contain a key with the name that is expected to be of an array type can also be controlled by options in the query language of the underlying document database. An example preserving documents in the result that do not have a key `address` at all is shown in listing 2.8. These documents will be returned by Mongo DB with a key `address` added which gets a value of `NULL` for this key.

Listing 2.8: Customer Source from a Document DB preserving documents missing an address.

```
SOURCE CustAddWithPosAndEmpty TYPE LIST QUERY ROLES
                              (_id: IDENTIFIER,
                               adrPos: IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.aggregate([ { $unwind:
    { path : "$address",
      includeArrayIndex : "adrPos",
      preserveNullAndEmptyArrays: true } } ])
END
```

Note that it is not required to flatten the arrays in documents from a document DB. Sources may still contain keys of array types, but the Checks and Actions of the DSL explained below do not provide specific array handling capabilities apart from an option to pass an array value to a parameter expecting a `LIST` value. As discussed above, we prefer to use the weakly typed approach in our DSL and leave the type-specific operations to the source database systems which are already capable of dealing with these specific types. Many of the potential array operations that might be needed can be implemented with the `UNNEST` query option as explained above in combination with the position information and our Checks language (see examples in section A). An easier to use syntax for such operations might be possible if integrating them directly into the DSL for convenience; this is left for future work in the current project.

Handling of the extended type system regarding sub-documents in document databases is not an issue for the source declaration as they are processed as is and added to the respective sources. Thus, their handling and access will be discussed below when the check's and action's details are introduced.

2.2.4. Checks

Checks are the skeletons of data quality rules, they might be compared to functions/methods from general purpose programming languages. A Check is basically

a method for data quality control. They always include RETURN statements, which yield detected quality problems. Thus the overall result is a list of quality problems that will later be evaluated by the Action.

Listing 2.9: DSL code for a NULL check.

```
CHECK NotNullCheck ON value:
  RETURN value IS NULL;
END
```

Listing 2.9 shows one of the most basic Checks, a check for NULL values. The second listing 2.11 compares two sets and checks whether they have the same amount of entries. Lastly listing 2.12 shows a check for distribution matches. Going back to the types of inaccurate data described in section 1.1, the three shown Checks cover all introduced categories. The NullCheck stands for the most basic *column property analysis*. Due to its nature the QuantityCheck covers *structure analysis* as well as *simple and complex data rule analysis* depending on what parameters are used. Lastly the DistributionCheck performs a *value rule analysis*.

To cover all these categories without creating a too complex language the expressiveness is three-tiered. First, there are boolean or mathematical expressions described directly by the language in combination of variable usage, IF statements and FOR loops as known from normal programming languages. Second, there is an SQL-like language to process data sets using typical operations like projection, selection, join and grouping. Third, there are built-in methods to hide complexity, like a method to check multidimensional cube data.

Accessing sources within the SQL-like statements in the checks works straightforward for relational database systems as sources. The name of a source can be used in the FROM part to declare the source data to be checked. Individual columns within such sources are identified by their attribute names a_i which are results from the source definition in other parts of the SQL-like statements. Attributes are assumed to be of non-complex data types.

Similarly, key names k_i are used to identify individual attributes of document objects for sources originating in document database systems. For simple data types this is similar to relational data sources. If a key k_i is not present in any of the documents, this document is excluded from consideration in this check. If key k_i is of a document data type itself, the classical dot notation may be used to access key-value pairs within such a document. As documents may be nested hierarchically, path expressions separated by dots may be used to navigate into the hierarchy. Note though, that non-existence of any of the keys in such a path expression results in the document being excluded from consideration in the current check. This is also true, if any key in any of the path expressions within a single check is not existing. However, the NOT NULL Check has a specific semantics where missing elements are considered to have a value of NULL.

Note though, that in most cases the value specification using dot notation will rather be used in the Action part (see below), because the Checks themselves are often generic and operate on role defined values or lists that are assigned in the Action part.

A Source list element is excluded from the current Check or Action, if a key is

actually present within a document, but does have a data type that does not match the syntax used to access information for this key (e. g. the inside of an array cannot be accessed as a sub-document and vice versa). Nevertheless, a whole array value can be used by simply specifiying the array's key name. The array elements are automatically treated as anonymous `LIST` values in the Check in this case.

Even though array handling is not included in the DSL at this time, many important functions can still be realized in the DSL by using the `UNNEST` feature of the document database query language with position selection as explained in section 2.2.3. In combination with a `GROUP BY` operation in the Check, the whole array can be reconstructed in the Check as long as the DSL provides an aggregation function necessary to assemble single array elements together into the array. This is illustrated in figure 2.10 below assuming that we have a String concatenation function and the array elements are strings.

Listing 2.10: DSL code for reconstructing arrays by grouping and aggregation.

```
CHECK PhoneNumberListCheck
LIST phoneNos :=
  SELECT _id, name, STRING_CONCAT(phone)
  FROM CustomerPhoneProj
  ORDER BY phonePos
  GROUP BY _id, name;
  ...
```

Listing 2.11: DSL code for quantity checking.

```
CHECK QuantityCheck ON LIST list1 WITH LIST list2:
  LIST cnt1 := SELECT COUNT(*) FROM list1;
  LIST cnt2 := SELECT COUNT(*) FROM list2;
  RETURN cnt1 != cnt2;
END
```

The idea behind Checks is - analogous to Sources - to cover technical details behind a structure in order to provide a domain expert with tools to describe data quality rules. The most basic Checks and some complex analysis will be predefined in the rule catalog. The goal is to provide a substantial Check foundation for the user in order to cover most needs. If the demand for individual Checks arises nonetheless, user defined Checks may be composed as well. Due to the nature of the technicality of these Checks, even though we will provide a language to compose them, this task will most likely be assigned to technical users.

Listing 2.12: DSL code for distribution checking.

```
CHECK DistributionCheck ON LIST data(val)
      WITH refDistribution(from, to, bin, perc):
  LIST cnt := SELECT COUNT(*) FROM data;
  LIST bindata := SELECT VAL2BIN(val, refDistribution) ROLE
     ↪  bval
```

```
              FROM data;
  LIST bincnt := SELECT bval ROLE bval, COUNT(*) ROLE acnt
             FROM bindata
             GROUP BY bval;
  LIST hist :=
    SELECT bin, perc, NVL(acnt, 0) ROLE acnt
    FROM bincnt RIGHT JOIN refDistribution ON bval = bin;
  LIST score1 :=
    SELECT (acnt - cnt * perc) * (acnt - cnt * perc) /
           (cnt * perc) ROLE cell
    FROM hist;
  LIST score :=
    SELECT SUM(cell) ROLE score FROM score1;
  RETURN LIST score;
END
```

Since Checks are just archetypes of data quality rules, they provide parameters to utilize them properly. There are two kind of parameters for Checks, the first parameter provides the information which data has to be checked for data quality issues. The second type of parameters are control parameters, they are used to aid in checking. For example the QuantityCheck expects two parameters, the first being a list of values (e.g. the contents of a customer table from a DWH), whose quantity should be checked against the amount of values of the second parameter, another list of values (e.g. the content of the original customer table). These parameters are divided via the Keywords `ON` for "data-to-be-checked"- and `WITH` for control-parameters. Whereas each Check contains exactly one `ON` parameter, which only accepts a Source as input, the amount of `WITH` parameter is not limited. Additionally, it is possible to also pass arbitrary values via a `WITH` parameter.

Returns

The ultimate goal of checks is to detect data quality issues, in order to present these issues the `RETURN` statement is used. There a few conventions of what will happen when different types of data is returned. Due to the three kinds of DQ-issues defined in section 2.2.7, different information is needed for each type. The first type needs a score, an identifier record and optionally a value record. Type 2. errors need a score and the offending value. The last kind only needs a score. With this information we define the behavior of return values as follows:

Boolean: If a record with a single boolean value is returned and this boolean value evaluates to true the current `ON` parameter is added to the return-value pool with a score of 0 (score of 0 is the default value, which results in an error if no threshold is given).

Double: If a record with a single double value is returned the current `ON` parameter is added to the return-value pool with the double value as the score.

Record: If a record with more than one value is returned it is assumed that it contains at least a score value and optional value and identifier fields. These fields will be extracted and added to the return-value pool.

All returned values are bundled and will later be evaluated by the result in part of the action.

2.2.5. Actions

With Sources and Checks defining the *what should be checked* and the *what quality aspects should be checked for* respectively, Actions are responsible for two things. Firstly, they connect Sources and Checks resulting in data quality rules. Secondly, they deal with the result and therefore conclude data quality rules. Actions were designed with natural language in mind, so that a rule composed in the DSL reads as a sentence which gets its point across without knowing the language itself. Following are a few examples of Actions which cover different groups of inaccurate data.

Listing 2.13: DSL code for an example of a *column property analysis*.

```
ACTION NotNullCheckCustomerName:
  EXECUTE NotNullCheck ON EACH Customer(name)
  RESULT IN ERROR
END
```

Listing 2.13 shows an Action testing the customer names for null values. The NullCheck is designed for single values, because there is no `LIST` keyword in front of the parameter in the check. Thus the action automatically calls the check for each item of the data source individually. This data quality rule falls under the category of *column property analysis*.

Listing 2.14: DSL code for an example of a *structural analysis*.

```
ACTION DWHQuantityCheckCustomer:
  EXECUTE QuantityCheck ON DWH_Customer WITH Customer
  RESULT IN ERROR
END
```

Shown in listing 2.14 is an example for a check of the result of an ETL process, it checks whether the amount of tuples in the DWH fits the original table. This data quality rule falls under the category of *structural analysis*.

Listing 2.15: DSL code for an example of a *value rule analysis*

```
ACTION AgeDistributionCheckCustomer:
  EXECUTE DistributionCheck ON CustomerWithAge(age)
  WITH AgeHistogram
```

```
  RESULT IN WARNING ABOVE 14.07 AND ERROR ABOVE 18.48
END
```

A more complex example is shown in listing 2.15. It goes back to the example introduced in section 2.2.4 and uses the customers dates of birth and a reference age distribution to compare the current customer ages with the saved reference distribution. The result is a score, which is evaluated by the `RESULT IN` part of the Action. This data quality rule falls under the category of *value rule analysis*.

The notation to access parts of a source definition follows the same logic as described for the Checks previously:

- Attributes of relational sources or key-value pairs of document sources with non-complex data types are accessed by specifying the Source name with the corresponding attribute or key name in parenthesis.
- Key-value pairs in document sources refering to array values can be used as a whole as anonymous list values by using the key's name inside the Source name.
- Key-value pairs in nested documents within a document can be accessed by using the Source name together with a path expression. Listing 2.16 shows how a predefined pattern Check is applied to a nested value. Every element of the path belongs to one level of nesting of documents describing the key's name that carries the sub-document as value. The example in listing 2.17 shows a `NOT ↪ NULL` Check on the `name` key inside the `street` sub-document of the `address` sub-document of the Source `CustomerAddress`.

Listing 2.16: Using an existing Check on a nested value.

```
ACTION CustAddressZipCheck:
  EXECUTE PatternCheck
    ON EACH CustomerAddress(address.zip: value)
    WITH ZipPattern
  RESULT IN ERROR
END
```

Listing 2.17: Navigating into a nested document originating from a Document DB.

```
ACTION NotNullCheckCustomerCountry:
  EXECUTE NotNullCheck
  ON EACH CustomerAddress(address.country: value)
  RESULT IN ERROR
END
```

2.2.6. Roles

Within Check parameters, the mapping of attributes is important. For example, the distributionCheck tests the distribution of values in a column called `val`. However, in

the source to be tested, the column might be named `age`. Thus, we need to map the column age to val. We call this mapping the `role` mapping, because the column age plays the role of the val part within the check.

It can happen that a role must be mapped to a different number of attributes for different use cases. As an example, look at the following check that tests referential integrity:

Listing 2.18: DSL code for referential integrity.

```
CHECK ReferentialIntegrityCheck ON LIST src(fk)
                                WITH LIST target(pk):
  RETURN LIST
    SELECT fk ROLE value
    FROM src LEFT JOIN target ON fk = pk
    WHERE pk IS NULL;
END
```

In this check, we have two roles: `fk` and `pk`, denoting the foreign key to be checked and the referenced primary key. However, the check might be called to test foreign key relationships with compound keys. Thus the roles fk and pk must be bound to multiple attributes of the data source. See for example the following two data sources:

Listing 2.19: Source for work steps.

```
SOURCE Workstep TYPE LIST QUERY
                ROLES (prjid: IDENTIFIER,
                       stepid: IDENTIFIER):
DATABASE projects NATIVE
  SELECT prjid, stepid, name, description
  FROM workstep
END
```

Listing 2.20: Source for works in relation.

```
SOURCE Worksin TYPE LIST QUERY
               ROLES (empid: IDENTIFIER,
                      prjid: IDENTIFIER,
                      stepid: IDENTIFIER):
DATABASE projects NATIVE
  SELECT empid, prjid, stepid FROM works_in
END
```

In this case, a compound primary key is referenced by a compound foreign key. So we call the check in the following way:

Listing 2.21: Source for works in relation.

```
ACTION WorksRefInt:
```

```
  EXECUTE ReferentialIntegrityCheck
       ON Worksin(prjid: FK, stepid: FK)
     WITH Workstep(prjid: PK, stepid: PK)
  RESULT IN ERROR
END
```

Thus, all roles must be bound to a list of attributes when the check is called. The order does matter, binding the attributes of Worksin in an order not matching the order of Workstep would result in wrong comparisons.

Roles can either be bound when the check is called or when a source is defined. However, normally only the default role IDENTIFIER is bound for every source. Furthermore, each attribute of a source is bound to role with the same name. Later bindings override earlier bindings, see the following example:

Listing 2.22: Example for role bindings: Source Customer.

```
SOURCE Customer TYPE LIST QUERY ROLES (id: IDENTIFIER):
DATABASE cust NATIVE
  SELECT id, name, firstname, dob, email,
               title, bonus
  FROM customer
END
```

Listing 2.23: Example for role bindings: NameCheck.

```
CHECK NameCheck ON tab(name, title):
  RETURN name IS NULL;
  RETURN NOT (title IN {"Mr", "Mrs", "Ms"});
END
```

Listing 2.24: Example for role bindings: Action.

```
ACTION NameCheckCustomer:
  EXECUTE NameCheck ON EACH Customer(name: name,
                                     firstname: name,
                                     title: title)
  RESULT IN ERROR
END
```

In this example, first the Customer source is defined. It consists of records of size four with the roles assigned to the columns as shown in table 2.1a.

When the check is called, the roles `name` and `title` are defined. As they were already defined, the old definition is overwritten. This leads to the role assignment as shown in table 2.1b.

It would also be possible to call the Check with the following code:

```
EXECUTE NameCheck ON Customer(name: name, firstname: name)
```

1	2	3	4
name	firstname	title	id IDENTIFIER

(a) Role assignment of the Source

1	2	3	4
name	firstname name	title	id IDENTIFIER

(b) Role assignment when calling the Check

1	2	3	4
	firstname name	title	id IDENTIFIER

(c) Wrong role assignment

Table 2.1.: Role example

However, leaving out the part `name: name` is not possible, as the redefiniton of a role replaces the whole role. Thus calling

```
EXECUTE NameCheck ON EACH Customer(firstname: name)
```

would result in the assignment shown in table 2.1c, which is most likely incorrect. In order to further simplify role assignments, a few special rules hold:

1. When a parameter of a check has no role declaration part, a single role is assumed that has the same name as the parameter itself. Thus, the parameter declaration `value` is implicitly interpreted as `value(value)`.
2. When an actual parameter in the check call lists roles without assigning new roles, the roles are assigned implicitly to the roles listed in the check body. In this case, the number of required roles has to exactly match the number of given roles.
3. When an actual parameter in the check does not list any roles, and the called check only requires a single role for this parameter, the role is implicitly defined to span all attributes of the actual parameter value.

A few examples illustrate these rules. First, look at the following check:

Listing 2.25: Example for role rule 1 (check).

```
CHECK NotNullCheck ON value:
  RETURN value IS NULL;
END
```

Listing 2.26: Example for role rule 1 (action).

```
ACTION NotNullCheckCustomerName:
  EXECUTE NotNullCheck ON EACH Customer(name)
  RESULT IN ERROR
END
```

Here, first rule 1 interprets the check parameter `value` as a parameter with a single role `value`. Furthermore, the call with parameter `Customer(name)` is extended due to role 2 to `Customer(name: value)`. Overall, the verbose form would be:

Listing 2.27: Expanded role example (check).

```
CHECK NotNullCheckVerbose ON value(value):
  RETURN value IS NULL;
END
```

Listing 2.28: Expanded role example (action).

```
ACTION NotNullCheckCustomerNameVerbose:
  EXECUTE NotNullCheckVerbose ON EACH Customer(name: value)
  RESULT IN ERROR
END
```

Rule 3 is typically used when constant sources are used as actual parameters. For example, when the email pattern is used as parameter value, a role that spans the only existing attribute of the pattern is implicitly defined. This is shown in the following example:

Listing 2.29: Example for parameter without role (check).

```
CHECK PatternCheck ON value WITH pat:
  RETURN NOT MATCHES(value, pat);
END
```

Listing 2.30: Example for parameter without role (action).

```
ACTION PatternCheckCustomer:
  EXECUTE PatternCheck
  ON EACH Customer(email) WITH EmailPattern
  RESULT IN ERROR
END
```

2.2.7. Return values

A return statement within a Check returns quality problems. However, it does *not* finish the execution of the Check. So multiple return statements are possible and

useful. Or a return statement can be issued with a loop each time a quality problem is detected.

For proper storage and evaluation of the found quality problems, a unified format to identify problems is necessary. In order to define such a format, we need to classify quality problem according to the unit of data that is related to the problem:

1. Problems that are related to individual data records. This includes also all problems that are related to individual column values like NULL values or range checks.
2. Problems that are related to values that occur in multiple records. For example, a uniqueness test might identify a group of three records sharing the same value in some columns that are supposed to be unique.
3. Problems that are related to a whole data source. An example is a skewed age distribution in a customer table that cannot be pinned to individual records.

All problems are returned as data records containing a problem description. We require these records to define some standardized roles for unified analysis of the results. The following roles are defined:

- `Check` The name of the Check that identified the problem
- `Action` The name of the Action that identified the problem
- `Source` The data source that contains the problem
- `Identifier` When the problem is of type 1, this identifies the offending record
- `Value` The value[1] that identifies the problem (only for type 1 or type 2 problems)
- `Score` Severity of the problem, either 1 when there is no differentiation between severities, or a number that will be larger for higher severity problems.

In order to avoid filling all these roles individually, the return statement contains some default logic that helps to fill these roles:

- The role Check will always be set automatically by the innermost Check that detected the quality problem. This means that an outer Check that returns problems found be inner Checks will not modify the Check role.
- The role Source will also be set automatically matching the Source that was used as an actual parameter for the ON parameter. This means that the Source name must be tracked when a Check calls another Check.

[1] value in this context is a record, therefore it may contain multiple atomic values

- When the return statement includes a boolean expression, the expression is evaluated. In case it evaluates to TRUE a problem record is returned. Otherwise, no record is returned from this statement. In case the ON parameter is a single record, the following values are automatically defined: Identifier, Value and Source (from the parameter). When the ON parameter is a SET parameter, only the role Source is defined, as it is assumed that the Check has tested the source as a whole.

- When the return statement includes a double expression, the logic is basically identical to the previous case. However, the value of the expression is interpreted as a score for the severity. In this case, the warning and error bounds of the calling Action are used to assess whether the problem should be reported. Only when the severity is above the warning threshold, a return value is generated. Furthermore, the value is included as the Score role in the return record.

- When the return statement includes a set expression, each record of the result set is required to define the roles Value and Score, if appropriate for the problem. The role Identifier is set automatically when available. To define these roles, the SELECT part of SQL expressions contains a special role syntax shown below: `SELECT` <*expr*> `ROLE score,` <*expr*> `ROLE values, ... FROM ...` This is basically the same as Alias names in standard SQL, with the exception that multiple columns can be assigned to the same role:
  `SELECT` <*expr*> `ROLE values,` <*expr*> `ROLE values, ... FROM ...`

2.3. DSL Description

2.3.1. Expressions

Expressions are like normal expressions in typical programming languages, except that they always operate on records and not on simple values. However, they can behave in the same way when the records only have a single attribute.

The basic building blocks for expressions are constants and variables. Constants are denoted in the following syntax:

Listing 2.31: Syntax for a constant.

```
constval ::=
  '[' atomval ( ',' atomval )* ']' | atomval
atomval ::=
  <stringconst> | <numericconst> |
  TRUE | FALSE |
  NULL
```

This means, that a constant is either a record in square brackets or a single constant value. A single constant value is treated as a record with a single attribute, thus the constants `13.5` and `[13.5]` are the same. String constants are enclosed with double

quotes ("). A double double quote escapes a double quote within a string. Numeric constants use a single dot as a decimal delimiter. So the following expressions are all valid constants:

Listing 2.32: Examples for constants.

```
[3, "Hello", TRUE, 4.8]
[TRUE,NULL,17]
FALSE
"Hello World"
"Hello ""World"""
```

The next building blocks are values. Values can either be specified by identifiers (i.e. variables). The variable's value will be replaced for the identifier when the expression is evaluated. Furthermore, a value can be created by calling a method with a list of actual parameters, or by calling one of the predefined aggregate functions on some expression. Thus we have the following syntax for values:

Listing 2.33: Syntax for values.

```
val ::=
  constval | varval | callval | aggval
varval ::=
  <name>
```

The provided built-in methods (*callval* and *aggval*) are described in the subsection 2.3.3. The value of a variable might either be a single records or a list of records. When a list variable is encountered where a normal record is expected, the list length is checked. When the length is exactly 1, then the list is implicitly converted to a single record. When the length is either 0 or larger than 1, a runtime-error is thrown. An array-like syntax with square brackets is provided to fetch a specific element from a list of records, as shown in listing 2.34.

A value that is used within an expression might carry role information. This stems from the list where the variable originated. A subrecord containing all the attributes assigned to a role might be selected using the dot-notation shown in *exprval*. Furthermore, expression can contain sub-expression using parenthesis as usual to overwrite the standard operator precedence.

Listing 2.34: Syntax for expression values.

```
exprval ::= ( '(' expr ')' | val )
            ( '[' expr ']' )? ( '.' <role> )?
```

In the next step, expression values can be compound to terms using binary operators, comparison operators:

Listing 2.35: Syntax for expressions.

```
expr ::= expror | listexpr
```

```
expror ::= expror OR exprand | exprand
exprand ::= exprand AND exprrel | exprrel
exprrel ::= exprterm relop exprterm | exprterm
relop ::= ( '<' | '<=' | '==' | '!=' | '>=' | '>' )
exprterm ::= exprterm ( '+' | '-' ) exprfactor | exprfactor
exprfactor ::= exprfactor ( '*' | '/' ) exprnot | exprnot
exprnot ::= NOT exprnot | exprun
exprun ::= exprval |
           exprval IS NULL |
           exprval IN listexpr |
           '-' exprval
```

As expressions always operate on records, semantics for the arithmetic, comparison and boolean operators have to be defined. Arithmetic and comparison operators are applied element-wise to the operands. Thus the operands must match in size. When one operator has more than one element, and the other operand has only a single element, this element is used as a counterpart for all elements of the other operand. Some examples for this behavior:

Listing 2.36: Examples for operators.

```
[2,5] + [4,7] yields [6,12]
[2,3,4] + 1 yields [3,4,5]
[2,3,4] + [1] yields [3,4,5]
[2,3] + [4,5,6] yields a runtime error
```

Type conversion is done implicitly. The operators can only work on operands of compatible types. NULL values are treated to have no type; and any operation on NULL values yields NULL again. The operators are evaluated according to the datatypes. Arithmetic operators are applicable only to numeric values; comparison operators only to strings and numeric values; plus is also applicable to strings meaning concatenation; and for boolean operands, only equality or inequality is applicable. The following examples show these semantics:

Listing 2.37: Examples for operators (2).

```
NULL + 17 yields NULL
"hallo" + 10 yields a runtime error
"hallo" + TRUE yields a runtime error
16 <= 17 yields TRUE
FALSE < 1 yields a runtime error
"hallo" < "zebra" yields TRUE
"180" < "19" yields TRUE (lexicographic)
TRUE = FALSE yields FALSE
"hallo" - 19 yields a runtime error
TRUE * FALSE yields a runtime error
```

The Boolean operators AND, OR and NOT behave differently. They reduce the left-side and right-side records beforehand to single Boolean values. Here, only records with only TRUE values are reduced to TRUE, all other records (including FALSE or NULL value) reduce to FALSE. A record with non-Boolean components yields a runtime error.

Again, some examples illustrate this behavior:

Listing 2.38: Examples for Boolean operators.

```
[TRUE,FALSE] yields [FALSE]
[3,4] <= 4 yields [TRUE]
[3,4] <= 3 yields [FALSE]
[3,4] > 3 yields [FALSE]
[TRUE,FALSE] AND [TRUE,TRUE] yields [FALSE]
[FALSE,FALSE] OR [TRUE,TRUE] yields [TRUE]
[TRUE,NULL] yields [FALSE]
NOT [1,0] yields a runtime error
NOT [TRUE,FALSE] yields [TRUE]
```

The reduce logic is applied either when Boolean operators are applied, or when an expression is evaluated in a context that requires a Boolean result value (i.e. in the condition of an IF statement or in the WHERE part of a list expression). One consequence of this logic is that the `IS NULL` operator will only yield `TRUE` if *all* attributes of a record are NULL:

Listing 2.39: Examples for NULL test.

```
[4,NULL] IS NULL = [FALSE,TRUE] ==> [FALSE]
[NULL,NULL] IS NULL = [TRUE,TRUE] ==> [TRUE]
```

2.3.2. List expressions

List expressions are expressions that yield a list of records. They are comparable to relational SQL statements, although the special treatment of roles and list variables allows these expressions to be parameterized in a very flexible way, including the exchange of tables and column.

On the top layer, a list expression can either be a statement similar to SQL statements, or a method returning a list, a variable name, or a constant list enclosed in braces:

Listing 2.40: Syntax for a list expression.

```
lstexpr ::= lstsel | lstmethod | <var> | '{' constlst '}'
```

List methods are described in subsection 2.3.3. A constant list is simply a list of constant values:

Listing 2.41: Syntax for a constant list.

```
constlst ::= constval ( ',' constval )*
```

SELECT list expressions use a syntax that is close to SQL, however with some important differences that are described in the following:

Listing 2.42: Syntax for SELECT list expressions.

```
lstsel ::=
  SELECT lstselect
  FROM lstfrom
  ( WHERE expr )?
  ( GROUP BY lstgroupby
    ( HAVING expr )? )?
  ( ORDER BY <role> ( ',' <role> )*
```

The select part is simply a list of expressions, where each expression can be assigned to a role. Alternatively, a star can be used to select all available attributes from the result.

Listing 2.43: Syntax for the select part.

```
lstselect ::=
  expr ( ROLE <role> )? ( ',' expr ( ROLE <role> )? )* |
  '*'
```

In the FROM part, a list of variables can be specified, with JOIN clauses defining how to combine these sources. The variables must either refer to lists or (via parameters) to data sources. Instead of variables, also sub-list-expressions enclosed in parenthesis (like subqueries in SQL) are allowed. All types of outer joins are supported, however, we only support equi-joins.

Listing 2.44: Syntax for the from part.

```
lstfrom ::=
  lstfromsrc ( lstjoin lstfromsrc ( ON <name> '=' <name> )?
    ↪  )*
lstfromsrc ::=
  <varname> | '(' lstexpr ')'
lstjoin ::=
  ( LEFT | RIGHT | FULL )? JOIN
```

In the GROUP BY part, a list of roles can be specified. The data in the attributes assigned to these roles will be used to group the result. The roles are either existing roles or those assigned in the SELECT part. This slightly differs from SQL, where you cannot access alias names from the SELECT clause in the GROUP BY. On the contrary, we do not allow arbitrary expressions here, we only allow to use those expression that where defined in the SELECT clause.

Listing 2.45: Syntax for the group by part.

```
lstgroupby ::=
  <role> ( ',' <role> )*
```

When a GROUP BY is used, every role in the SELECT part must either be the result of an aggregation function or must be listed in the GROUP BY clause.

Expressions in the SELECT, WHERE and HAVING part are always executed in the context of a single record that is currently processed. This record might have roles. Role names of these roles are also allowed as variable names in the expressions. As they share a common name space, scoping semantics have to be defined. We apply a simple rule: variable names always hide role names. Let's look at an example:

Listing 2.46: List expression example.

```
minval := 10;
LIST res :=
  SELECT value, identifier
  FROM data
  WHERE value > minval;
```

Assume that data is bound to a data source that provides the roles value, identifier and minval. In this case, the reference to minval in the code references to the variable minval defined directly before the list expression, and not to the role minval from data. Now look at this code snippet:

Listing 2.47: List expression example 2.

```
minval := 10;
LIST res :=
  SELECT *
  FROM data JOIN (SELECT value ROLE minval FROM param)
  WHERE value >= minval;
```

Due to our rule, the reference to minval in the WHERE clause still refers to the variable minval instead of the role minval defined by the sub-list-expression. This might be surprising, however the advantage are clear semantics. Compare with the following example:

Listing 2.48: List expression example 3.

```
LIST sub := SELECT value ROLE minval FROM param;
minval := 10;
LIST res :=
  SELECT *
  FROM data JOIN sub
  WHERE value >= minval;
```

In this case, again, the variable is substituted. However, from a user's point of view, it would be best not to use the same names for roles and variables. Furthermore, variables are substituted immediately. Look at the following example:

Listing 2.49: List expression example 4.

```
minval := 10;
LIST res :=
  SELECT *
  FROM data JOIN sub
  WHERE value >= minval;
minval := 20;
RETURN LIST SELECT * FROM res;
```

Here, minval is substituted with 10 immediatly in the second statement.

2.3.3. Built-in Methods

Currently, the following methods within expressions are built-in that return records:

Listing 2.50: Built-in methods in expressions.

```
callval ::=
  ABS '(' expr ')' |
  NVL '(' expr ',' expr ')' |
  BINOM '(' expr ',' expr ',' expr ')' |
  LENGTH '(' listexpr ')' |
  MATCHES '(' expr ',' expr ')' |
  VAL2BIN '(' expr ',' listexpr ')' |
  DOB2AGE '(' expr ')'
```

- `ABS(x)` returns the absolute value of the parameter `x`. If the parameter is a record, the absolute value is calulated element-wise.
- `NVL(x,y)` replaces each NULL-element of the record `x` with the corresponding value in the record `y`.
- `BINOM(x,n,p)` returns $P(X \leq x)$ for $X \sim Binom(n,p)$, where x is the first parameter. Each parameter must be a single value (when a record is used, only the first element is used).
- `LENGTH(l)` returns the number of elements of the list `l`.
- `MATCHES(value, pattern)` tries to match `value` against the regular expression given in `pattern`. If there is a match, `TRUE` is returned, else `FALSE`.
- `VAL2BIN(value, bins)`: Here, `bins` must be a histogram definition, i.e. a list that contains the roles `from`, `to`, and `bin`. The `value` is checked agains each

element. If `value` is between `from` (inclusive) and `to` (excluive), then `bin` is returned. If no matching bin is found, then NULL is returned. If the upper bound `to` is NULL for a bin, then only the lower bound is tested. The resulting record has a single role `name`.

- `DOB2AGE(dob)`: Converts a birth-date `dob` into an age. The resulting role is called `age`.

Aggregation methods are methods that are only valid in the context of the SELECT-part of a list expression. Otherwise, their usage will produce a runtime error.

Listing 2.51: Built-in aggregation methods.

```
aggval ::=
  MIN '(' expr ')' |
  MAX '(' expr ')' |
  SUM '(' expr ')' |
  AVG '(' expr ')' |
  COUNT '(' ( '*' | expr ) ')'
```

The logic of the methods should be clear from the name. The `COUNT` method counts either the overall number of records when called as `COUNT(*)` or the number of times the expression given as a parameter to the method is not NULL. `MIN` and `MAX` can be used for numeric and string expressions, while `SUM` and `AVG` only work for numeric expressions.

List methods are built-in methods that return a list. Currently, two methods are implemented:

Listing 2.52: Built-in list methods.

```
lstmethod ::=
  ARIMA '(' lstexpr ',' lstexpr ')' |
  CUBESCORE '(' lstexpr ',' lstexpr ')'
```

Details for these methods are given in chapter 3.

2.3.4. Sources

Sources are the interface to databases or other reference values the domain expert may work with. This Section gives a detailed description of the structure of sources and how they are syntactically made up as shown in listing 2.53.

Listing 2.53: Syntax for a Source

```
srcconstlst ::=
  SOURCE <name> TYPE LIST CONST rolelist ':'
    constlst
  END
```

```
srcconstval ::=
  SOURCE <name> TYPE CONST rolelist ':'
    constval
  END

srcquery ::=
  SOURCE <name> TYPE LIST QUERY roledecl ':'
  DATABASE <dbname> NATIVE
    <Query in the native language of the database>
  END

rolelist ::= ROLES '(' <role> ( ',' <role> )* ')'
roledecl ::= ROLES '(' roleasgn (',' roleasgn)* ')' )? ':'
roleasgn ::= <attr> ':' <role>
```

The given name identifies a Source, which later may be used as a parameter in Checks to either aid in checking for data quality issues when used as a `WITH` parameter, or it will be checked for problems when used as the `ON` parameter. The `TYPE` modifier in the header is used to define two properties, firstly it determines if the Source is a list of records or a single one, by optionally adding the keyword `LIST`. Secondly, it states whether this Source contains constant values or a query, in order to differentiate between Sources being a source data reference (`QUERY`) or constant values (`CONST`). An optional part of the header is the assignment of roles to attributes. The most common example is the declaration of primary keys via the identifier role. If a record entry does not get a role assigned, the default role will be the name of the indicator (in relational terms the column).

Listing 2.54: Example for a CONST Source representing an age distribution

```
SOURCE AgeHistogram TYPE LIST CONST ROLES (from, to, bin,
    ↪ perc):
 [0, 18, "0-18", .05],
[18, 25, "18-25", .20],
[25, 35, "25-35", .15],
[35, 50, "35-50", .25],
[50, 65, "50-65", .19],
[65, 80, "65-80", .10],
[80, 95, "80-95", .05],
[95, NULL, "95-", .01]
END
```

Roles for constant sources are applied as shown in listing 2.54, the idea is to name record entries sequentially, as they occur in the source, comma separated. Since constant values do not have an indicator in their definition, the assignment of `<attr> : < ↪ role>` can not be applied. In const sources, the role definition is mandatory, while in query sources, it can be omitted, resulting in the original attribute names used as

roles. When a role assignment is present in a query source, still all original attribute names are preserved as roles. This means you have to take care that there is no name clash between newly assigned roles and attribute names.

Listing 2.55: Example for a QUERY Source representing an employee table

```
SOURCE Employee TYPE LIST QUERY ROLES (id: IDENTIFIER):
DATABASE hr NATIVE
  SELECT name, dob, email, id FROM employee
END
```

2.3.5. Checks

Checks are methods for data quality monitoring. They range from simple null checks to complex time series or multi-dimensional analysis. Basically, every Check is a function that is called with some data to be checked and returns a set of quality problems found within the data.

The basic syntax to define a check is as follows:

Listing 2.56: Syntax for a check.

```
check ::=
  CHECK <name> ON formalparam ( WITH formalparam ( ',' 
      ↪ formalparam )+ )?:
    checkstmt*
  END
```

The name identifies the check. The ON parameter must be present an references the data to be checked. The WITH parameters are optional and can provide either steering parameters or other (reference) data used to check the ON parameter. The body of the check is simply a list of check statements that are introduced later.

A parameter is defined as follows:

Listing 2.57: Syntax for a formal parameter.

```
formalparam ::=
  LIST? <name> rolelist?
```

The LIST keyword means that the check expects a list of records for this parameter. The *rolelist* defines, which roles the check expects within records for this parameter. In case that the role list is omitted, a single role with the parameter name is expected. Thus, the parameter definition `street` is equivalent to `street(street)`. For details, see section 2.2.6.

The body of a check consists of a list of check statements. These can be a variable assignment, a RETURN statement or a control structure (IF or FOR), or a call to another check:

Listing 2.58: Syntax for a check statement.

```
checkstmt ::=
  checkasgn |
  checkreturn |
  checkfor |
  checkif |
  checkcall
```

A variable assignment can either assign a single record variable or a list variable, depending on the presence of the LIST keyword. The evaluation of list expressions is sometimes deferred by the internal optimizer. The EVAL keyword can be used to force immediate evaluation of the expression. Simple (record) expressions are always evaluated immediately.

Listing 2.59: Syntax for a variable assignment.

```
checkasgn ::=
  LIST <name> ':=' EVAL? lstexpr ';' |
  <name> ':=' expr ';'
```

A RETURN statement can either return all records from a list or a single record. The single record might be composed in place using a syntax similar to the SELECT part of list expressions. A RETURN statement does not end the execution of the check, it just adds records to the list of returned quality problems. Thus it is possible to call RETURN e.g. from inside a loop multiple times, emitting one quality problem each time.

Listing 2.60: Syntax to return quality problems.

```
checkreturn ::=
  RETURN LIST lstexpr ';' |
  RETURN checkretelem ( ',' checkretelem )* ';'

checkretelem ::=
  expr ( ROLE <role> )?
```

The control structures are defined as follows:

Listing 2.61: Syntax for an IF statement.

```
checkif ::=
  IF expr ':' checkstmt* ( ELSE checkstmt* )? END
```

The IF statement evaluates the expression to a boolean to decide, whether the IF or the ELSE part will be executed.

Listing 2.62: Syntax for a FOR statement.

```
checkfor ::=
  FOR <name> ':=' expr TO expr ':' checkstmt* END |
  FOR <name> IN lstexpr ':' checkstmt* END
```

The `FOR` loop comes in two flavors. The first variant is a numeric for loop the evaluates two expression to retrieve a lower and upper bound. The loop variable `<name>` is a numeric variable that runs from the lower bound to the upper bound (inclusive) with step length 1. The second flavor evaluates a list expression and runs the inner block for every element of the resulting list. Thus, `<name>` will be a record from this list in each iteration.

Finally, we can call another check from within a check:

Listing 2.63: Syntax to call a check.

```
checkcall ::=
  EXECUTE PERCENTAGE? <checkname> ON EACH? actualparam (
      ↪ WITH actualparam ( ',' actualparam )* )? (IF expr)?

actualparam ::=
  <name> roledecl? | constval
```

The check to be called is identified by its name. For the formal `ON` parameter and every `WITH` parameter, an actual parameter must be passed. The actual parameter can either be an identifier or a constant value. The identifier can either identify a variable or a source. Variable names mask the names of sources, as the system first looks for a matching variable, and if none is found, a matching source is looked up. If neither a variable nor a source is found, a runtime error is thrown.

A formal parameter might either expect a list or a record. The actual parameter passed must match this type with one exception: A list might be passed to a formal record `ON` parameter using the `EACH` keyword. In this case, the check is called multiple times, once for each element of the list. Any other mismatch will result in a runtime exception

Each formal parameter has a list of expected roles. The actual parameter must supply values for these roles. In order to match these roles, the original roles of the actual parameter might be redefined using a `roledecl`. We can have multiple cases:

- When the actual parameter value is a constant record, the number of elements of the record must match the number of roles of the formal parameter. The roles are assigned element-wise in order. No explicit role mapping is defined in this case.

- When the actual parameter value is either a source or a variable, it is expected to carry role information. The original roles must then be mapped to the roles of the formal parameter. There are multiple ways to specify this mapping. First, a full `roledecl` in the format `(orig1:formal1, orig2:formal2,...)` can be given. The order of the roles is not important in this case, and formal roles may be omitted. In case of omission, a formal role is expected to exist in the original roles, thus implicitly, a mapping `(formal3:formal3)` is assumed. An extreme case here is the omission of the whole `roledecl`. The other option is a `roledecl` in the form `(orig1, orig2, ...)`. In this case, the number of original roles must match the number of formal roles, and the mapping is defined by position.

Assume we have a formal parameter with the roles `(from, to, bin, value)`. We might have the following assignments:

- `[3, 10, "Children", 0.2]` is a constant value. The 3 will habe role `from`, 10 will have role `to`, etc.
- In the other cases the actual parameter is an `identifier` that must refer to a source or a variable with roles. We have the following subcases:
  - `identifier`: The roles `(from, to, bin, value)` are expected to exist in the source and will be directly used.
  - `identifier(end: to, start: from)`: The role `start` is mapped to `from`, and the role `end` is mapped to `to`. The roles `(bin, value)` are expected to exist in the source and will be directly used.
  - `identifier(start, end, binning, pval)`: This is equivalent to the mapping `identifier(start: from, end: to, binning: bin, pval: value ↪ )`.

When the actual parameter is passed to the check, only those roles that where defined in the formal parameter are accessible from the check. Thus, any other role that might have been present in the actual parameter is masked within the check.

When the roles for the ON parameter are mapped, an implicit mapping of the `IDENTIFIER` is added, so that the identifier of any source is always accessible in case it exists.

2.3.6. Actions

Actions apply Checks on Sources. This Sections gives a detailed explanation of how actions may be defined. Listing 2.64 shows the syntactic structure of an action.

Listing 2.64: Generic Action Definition

```
action ::=
  ACTION <name> ':'
    EXECUTE PERCENTAGE? <name>
    ON EACH? actionparam
    (WITH actionparam (',' actionparam)*)?
    actionresult
  END
```

The Action's header contains a name, so that the Action later may be grouped together with other Actions. An Action call consists of two parts, the Check call with its entailing parameter definition and the result evaluation. The calling of the check can look different depending on whether the `ON` parameter of that check expects a list of records or just a single record. If just a single record is expected, either the keyword `PERCENTAGE` or `EACH` have to be used, in order to decide whether the result should deal with the passed arguments in a overall fashion - meaning that a certain percentage of

records has to fail the Check in order to trigger an output, or apply the Check for each record individually.

Listing 2.65: Syntax for parameters used in Actions

```
actionparam ::=
  <name> ( '(' actionrole (',' actionrole)? ')' )?

actionrole ::= <role> | <name> : <role>
```

The role assignment of attributes of the parameter record is shown in listing 2.65. It can occur in two ways, the first one being implicit role assignment. The PersonCheck defines three roles for the `ON` parameter, if no role assignment is explicitly given in the Action, the roles will be assigned by order. The Action parameter `Customer(firstName, dob, email)` for the PersonCheck would implicitly map the roles as follows: `Customer(firstName : name, dob : birthDate and email : ↪ mail)`. The latter would be the second (explicit) type of role assignment. Assigning roles this way also enables the user to assign more than one attribute to the same rule.

Listing 2.66: Syntax for the evaluation of results

```
actionresult ::=
  RESULT IN WARNING ABOVE <number> AND ERROR ABOVE <number>
  | RESULT IN (ERROR|WARNING) (ABOVE <number>)?
```

The last part of an Action is the evaluation of the check outcome. As seen in listing 2.66, it begins with `RESULT IN`. Two kinds of result evaluation are possible, the first one being a simple true or false decision. An example would look like this: `RESULT ↪ IN ERROR`. This would mean that every record in the result set will be marked as a data quality problem. The second kind of result evaluation deals with score results, it enables the user to define score thresholds stating whether a score results in an actual data quality problem (`ERROR`) or just a suggestion of an issue (`WARNING ↪ `). This could look like the following: `RESULT IN WARNING ABOVE 2 AND ERROR ↪ ABOVE 3`. Since scores range from zero to infinity with higher numbers relating to more severe problems, the keyword `ABOVE` is used to describe the warning and error threshold.

Groups

In order to facilitate the execution of multiple rules, we introduced the grouping of Actions. Members of a Group can be either an Action or another Group. This enables the creation of a hierarchical structure representing the users Actions. Listing 2.67 shows the syntax for creating groups.

Listing 2.67: Syntax for grouping Actions and Groups

```
actiongroup ::= GROUP <name>:
```

```
((ACTION <name> | GROUP <name>)';')+
END
```

2.4. Environment

After introducing all three parts of the DSL, we want to give a conclusive description of what denotes a data quality rule. Like mentioned beforehand, our view of data quality rules includes three kinds of information. Since an Action unites Sources and Checks - and therefore includes them - and evaluates the result, it determines a data quality rule from our and a domain experts point of view. Besides that, some metadata related to the Action like execution and result histories are also part of a data quality rule, due to the continuous monitoring process our DQ system exerts for. Finally, we want to show two additional features of data quality rules we added to improve their usability.

In order to enable an easy way to manage rules we define an *active* flag to data quality rules. The composed rules will be executed by a scheduler (either one we provide or an external one already present in the clients system), due to the fact that these rules may run when either something relevant happened (e.g. a finished ETL process) or when the database has some spare time for queries so that important tasks are not interfered with. When a scheduler invokes a rule the active flag will be checked: if it is set, the rule will be executed. With this feature a domain expert may control which rules should be executed, without the need to interact with schedulers.

Another feature is the possibility to group rules together with added functionality of grouping groups, therefore the user may structure their rules hierarchically. As aforementioned, the rule execution will be triggered by a scheduler. In order to easily trigger the execution of many rules, the user will be able to create groups of rules which later will be initiated to execute by a scheduler. The active status specified beforehand, may also be set or revoked for a whole group of rules therefore allowing easy control over which rules are currently been evaluated.

2.5. Optimization

An important goal of our execution engine implementation is flexibility with respect to location of execution of check logic. In principle, checks can be executed in the quality engine itself or moved to the source databases by rewriting the source query. On the one hand, the execution engine should be able to execute all logic on its own, in cases where the target database does not have enough capabilities, either because of resource constraints or because of a lack of features. On the other hand, if possible, logic should be moved to the database for more efficient execution due to co-location with the data.

To reach this goal, we distinguish two cases. First, we have checks for individual records, that are called in a loop for each item of the source database. In this case, we extract what we call *safe conditions* from the Check, i.e. conditions that need

to be fulfilled for the Check to find quality problems. These conditions are then translated to filter conditions that are added to the native database query in the source database. This step needs individual code for each query dialect and might fail, if the target dialect does not support the relevant conditions. However, if it succeeds, we avoid fetching large numbers of records that will not produce output for the quality check anyway. As a very simple example, consider the `NotNullCheckCustomerName` in listing 2.13. From the check, we extract the condition `value IS NULL`. The role `value` is then back-substituted to `name` and the original query is modified to

```
SELECT * FROM (... original query ...) WHERE name IS NULL
```

For checks that work on `LIST`s, we follow a similar strategy. The first component is deferred execution. This means, that list expressions are evaluated only symbolically, i.e. variables are substituted with their current values. However, the list expression itself is not evaluated until it is actually needed (i. e. because it is part of a return statement or part of a condition that determines control flow). This leads to larger expressions that are build up part by part. The language still also has an `EVAL` statement that forces immediate evaluation of a list expression, if needed.

Upon final execution, the whole expression is optimized quite similarly to normal database optimizers. Our optimizer tries to push as much logic towards the data sources, where the logic is added to the original database query (similar to the above simple modification for `NotNullCheckCustomerName`). However, as soon as different source databases are involved in an evaluation, no further pushing of operations to the Sources is possible and the quality engine has to take over evaluation from this point on. For example, a join of two sources from different databases has to be processed internally in the data quality engine.

While the execution optimization has already provided promising results (cf. section 2.6), this part still remains an important issue for future improvements.

2.6. Evaluation

In this section, we look at the efficiency of our execution engine and especially at the effectiveness of our optimization approaches as performance will be a critical feature in e. g. large DWH quality use cases. For this, we use three different scenarios. First, we look at a simple NOT NULL check (NN), which is an example of a single record check. In this case, the optimizer modifies the source statement so that only critical records (i. e. NULL values) are selected. Second, we look at a referential integrity check (RI) which checks which records in one list do not reference valid records in the other list. This contains more complex list expressions that are translated to SQL by the optimizer in case of a relational data source. However, when the two tables are not located in the same database, this approach is limited. As a final, more complex, example of this list expression to SQL translation, we use a distribution check (Dist) which checks to which degree records in a source comply with a given reference distribution.

	Errors	0%			1%			10%		
	Size	1%	10%	100%	1%	10%	100%	1%	10%	100%
NN	Std.	0.25	1.57	16.45	0.22	1.35	18.15	0.21	1.36	17.22
	Optim	0.06	0.05	0.33	0.04	0.06	0.48	0.05	0.23	2.85
	SQL	0.01	0.04	0.53	0.01	0.04	0.53	0.01	0.12	1.85
RI	Std.	0.66	5.44	106.38	0.56	5.62	142.90	0.67	11.98	458.45
	Optim	0.15	0.79	16.38	0.12	0.84	14.11	0.11	1.28	16.74
	SQL	0.11	0.76	11.24	0.10	0.87	10.75	0.12	1.37	11.01
Dist	Std.	0.82	6.40	74.01						
	Optim	1.07	7.80	73.04						
	SQL	1.23	7.97	70.21						

Table 2.2.: Evaluation results (averages over 5 runs)

For evaluation, we use a person (approx. 5,000,000 records) and a plays relation (approx. 8,400,000 records) from a movie database. For each scenario, we look at different sizes of the data to check (1%, 10% and 100% of the original data). As the runtime might be influenced by the number of errors in the first two scenarios, we also use different numbers of errors (no errors, 1% errors, 10% errors). The distribution check calculates an overall score for the target relation taking each record into account, thus here we have no difference in effort related to error ratio. As benchmark for a lower limit, we also list runtimes of SQL statements running directly on the database that perform the same check. This is only possible as long as all data originates from a single database and also sacrifices all other benefits of *RADAR*as explained in section 2.1.

The results are summarized in table 2.2. For the NN check, we see that the optimized version performs much better than the standard version. The standard version suffers from a large number of calls to the check (for each person record) and is almost independent of error ratio. The optimized version is a significant improvement that is the better the smaller the error ratio (since fewer data has to be transferred to the engine). Also, the scaling behavior looks good as the runtime is roughly proportional to the data size. Runtimes for the optimized version are within the same order of magnitude as the raw SQL statement (that does not include any DSL processing). As the optimized version can only filter correct records, the runtime becomes larger with larger error ratio.

For the RI check in the standard version the scaling is not as good, both in terms of data size and error ratio. We assume that this is mainly due to our join implementation in the engine, which is currently not very efficient and subject to future improvements. However, the optimizer works quite well for this scenario again, reaching runtimes in the same order of magnitude as the raw SQL benchmark and similar scaling behaviour. Also, as expected, the runtime is almost agnostic of the error ratio in this case.

Finally, the Dist scenario shows a different picture. Here, the scaling works well, however the optimized version does not yield any improvement. A comparison with

the raw SQL statement shows, that this statement already consumes nearly all of the runtime so that no improvement is possible. A closer look revealed that this is due to the stored procedure we used to translate the `val2bin` method to SQL. A hand-crafted SQL version that avoids this procedure can be much faster, so we plan to modify our SQL generation to output this result in the future.

Overall, we can conclude that the optimizer works very well in general and establishes runtimes close to raw SQL statements that implement the same logic. However, as we cannot always count on it in cases when the underlying database has limited query capabilities (NoSQL) or when data from multiple sources is combined, we will also continue to improve the DSL interpreter. However, the efficiency improvements here are more tied to improvements in small details (e. g. join implementation, implementation of the expression evaluation) and not to general architectural issues. Thus the engine to execute *RADAR*can be considered sufficiently efficient.

3. Complex DQ Rules

The complex data quality rules in our system are different from simple rules in a few ways. First, they are not composed by data quality managers, rather than mined from past data in the later introduced data profiling step.

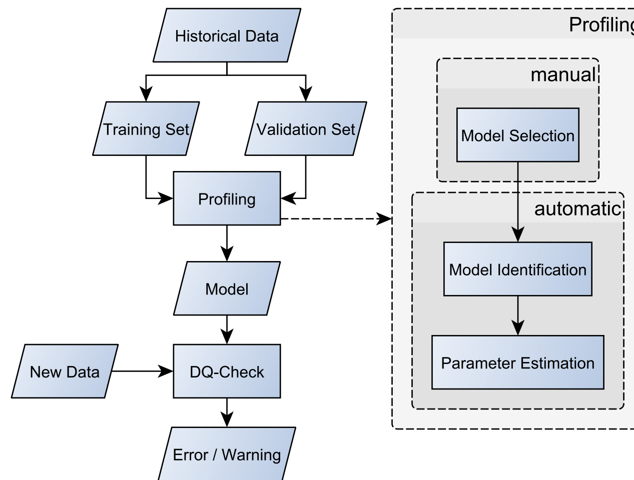

Figure 3.1.: Process flow of a complex data quality rule.

Figure 3.1 shows a generic process flow of a complex data quality rule. It starts with historical data which may be divided into training and validations sets. These sets are then processed by a given profiling method. Whereas the Model Selection (for time series analysis e.g. ARIMA) is up to the data quality manager, the model identification and parameter estimation are automated processes. Model identification consists of identifying the concrete model of the given class of models, i.e. AR(1) in the class of ARIMA models. Parameter estimation then determines the most appropriate parameters for the given model. With the knowledge gained by data profiling a model is created. This model is then used in a DQ-Check to test new data for anomalies (probable quality issues), which are then output to the domain expert for inspection.

3.1. Parametrization

Like previously said, models for the source data will be calculated by data profiling techniques. These models will then be used by Checks to test new data for quality issues. Meaning the models will be `WITH` parameters of Checks. Depending on the kind of Check, i.e. time series with ARIMA or multi-dimensional with a cube model, these parameters can be quiet long lists. In order to not inflate Actions to the border of illegibility, a sensible solution would be to wrap all parameters - belonging to a model - in a Source. A counterargument would be, that the parameters of some model might be human readable and understandable, so that a data quality manager might be able to manually adapt them. Condensed in a Source and simply passed to a Check via a variable name, a DQ manager would not be able to directly understand the logic behind the used model. Therefore, a careful consideration of how the model is introduced to the Check when designing more complex ones is needed. When it comes to ARIMA models for time series checks, a Source as parameter set is on the border of necessity, but we decided to use a Source nevertheless, since most models will have a parameter amount of about five to ten, which would result in a not easy comprehensible Action. Cube models definitely will need a source as a parameter set, due to the large amount of different parameters.

3.2. Univariate Time Series

In this section, we describe the treatment of time series models as data quality models for time series data. Most data is time-dependent and can thus be modeled as a time series. For example, sales data could be summed up to a daily sales volume and thus forms a univariate time series with one value per day. Other time intervals are possible as well, for example weekly or monthly aggregates.

The basic idea is to use a time series model to calculate an *expected* value for each item of the time series, using only the past values. The quality check can then compare the actual value of the time series with the expected value in order to find unusual values that might indicate quality problems. The magnitude of the difference between expected and actual value is the basis for the outlier score that describes the severity of the problem.

In the first iteration we only consider models of the ARIMA classes as models. This class of models is quite powerful and flexible and still easy to handle. The models are stochastic and thus can naturally model the variablility of the time series.

A time series in ARIMA types of models is described by the AR and MA part which stand for Auto Regressive and Moving Average respectively. The AR component is a linear combination of the previous values of the time series and a random component. The ϕ_p are the factors for the previous values and Z_t is the unpredictable error term:

$$Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + Z_t \tag{3.1}$$

The MA component is a linear combination of the previous random components Z_t:

$$Y_t = \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \cdots + \theta_q Z_{t-q} + Z_t \tag{3.2}$$

Combining both parts yields the following model:

$$Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \cdots + \theta_q Z_{t-q} + Z_t \quad (3.3)$$

Each Z_t is modeled using a normal distribution with mean 0 and variance σ^2.

The idea is to calculate an estimated value $\hat{Y}_t$ and compare that to the actual value Y_t. To calculate the estimated value (also called forecast value), the following formulae are used. For an AR time series, we simply set the random component to zero:

$$\hat{Y}_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} \quad (3.4)$$

The residual R_t is the difference between the expected and the acual value:

$$R_t = Y_t - \hat{Y}_t \quad (3.5)$$

For the forecasting of the MR part, the past residuals are used to estimate the past random components. This means, that Z_{t-1} is estimated as R_{t-1}. The current random value Z_t is estimated again as zero. This yields the following formula:

$$\hat{Y}_t = \theta_1 R_{t-1} + \theta_2 R_{t-2} + \cdots + \theta_q R_{t-q} \quad (3.6)$$

This definition is recursive, as the previous forecasts are needed to calculate the previous residuals, etc. Thus we need to store the old residuals, or recompute all of them and use these values when checking the current value of the time series.

The score for the value at time t is the absolute value of the residual normalized with the standard deviation of the random component:

$$S_t = \frac{|\hat{Y}_t - Y_t|}{\sigma} \quad (3.7)$$

3.2.1. Implementation

In this project we use the software package *R*[1] for more complex computations like the calculation and estimation of ARIMA models. *R*'s functionality is provided in packages, for our time series approach we use the `forecast` package, it provides functions for model identification, parameter estimation and eventually forecasting values. As mentioned earlier, we use this forecast to estimate how probable the corresponding time series value is. The forecast is used to calculate a residual, which then is divided by the standard deviation to yield a score, as shown in equation 3.7.

Listing 3.1: *R* code for generating a time series and computating a model.

```
library(forecast)
# Generates a time series with 100 data points
# of order 1,1,1 with an ar part of .75 and a ma part of
    ↪ .25
```

[1] see `https://www.r-project.org/`

```
timeseries <- arima.sim(list(order = c(1,1,1),
                             ar = 0.75, ma = 0.25), n =
                               ↪ 100)
# auto.arima is used for model identification
# and parameter estimation
model <- auto.arima(timeseries)
```

Listing 3.1 shows *R* code which generates an artificial time series and then computes a fitting model for the generated time series. Following examples will use the generated time series listed in Listing 3.2 and shown in Figure 3.2. It was generated with the parameters seen in Listing 3.1, therefore it roughly follows an order of (1, 1, 1) with an AR part of .75 and an MA of .25.

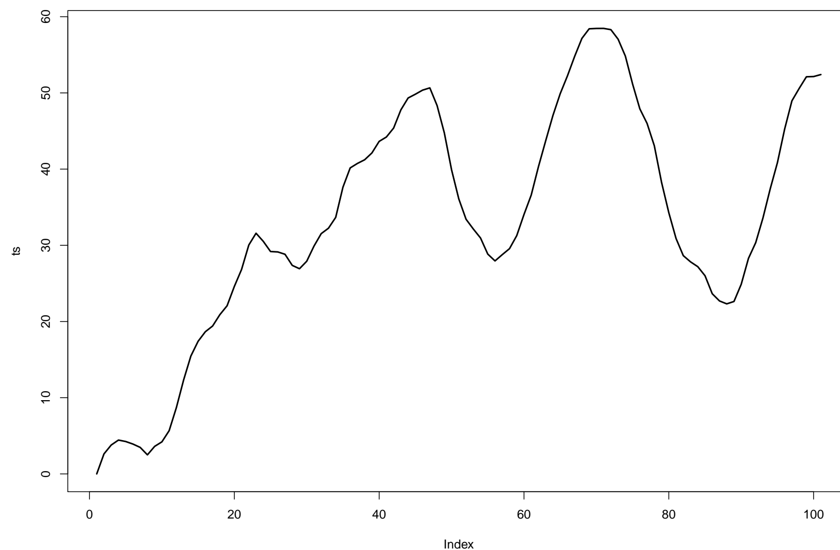

Figure 3.2.: Plot of the *R*-ARIMA Check with the sample time series.

Listing 3.2: Source representing an artifical generated time series.

```
SOURCE TimeSeries TYPE LIST CONST ROLES (value):
0.000000, 2.623743, 3.789620, 4.439828, 4.249843,
3.916899, 3.478510, 2.500894, 3.612794, 4.204696,
5.669945, 8.704310, 12.332818, 15.468202, 17.404085,
18.660676, 19.404775, 20.893627, 22.064165, 24.594094,
26.827290, 30.035758, 31.584607, 30.519828, 29.183136,
29.130115, 28.821930, 27.364103, 26.926484, 27.908534,
29.892975, 31.531580, 32.241380, 33.664232, 37.625139,
```

```
40.154500, 40.743571, 41.225403, 42.118486, 43.636868,
44.210719, 45.386753, 47.774890, 49.331057, 49.827593,
50.363815, 50.659891, 48.341681, 44.755109, 39.909498,
36.084458, 33.418436, 32.143091, 30.957857, 28.846263,
27.951211, 28.773923, 29.559525, 31.267978, 34.040998,
36.595921, 40.322161, 43.725928, 47.048210, 49.890612,
52.229083, 54.804125, 57.155170, 58.411904, 58.458849,
58.465982, 58.300552, 57.046101, 54.832691, 51.161610,
47.904541, 45.992418, 43.061357, 38.254406, 34.280723,
30.891375, 28.642376, 27.827763, 27.196900, 26.006698,
23.642374, 22.697840, 22.314777, 22.627406, 24.886459,
28.285309, 30.345886, 33.584469, 37.396241, 40.851070,
45.267623, 48.960385, 50.582479, 52.114948, 52.146251,
52.407847
END
```

When applied to a user selected time series, the `auto.arima` function is used to identify the model and estimate the corresponding parameters. While the object returned from the function contains many (17) fields, only a few are of interest to us. In order to apply the derived model to future time series data, we need to extract the the following attributes: order (arma), coefficients (coef) and variance (sigma2). With this information we can recreate the model, which will then be applied to the given time series. Listing 3.3 shows sample values of an ARIMA model of order 1,1,1. The arma attribute contains seven values which indicate the number of factors of the model as follows: AR, MA, seasonal AR, seasonal MA, period and number of seasonal and non-seasonal differences.

Listing 3.3: Sample values of a ARIMA time series model.

```
model$sigma2
# [1] 1.027177
model$arma
# [1] 1 1 0 0 1 1 0
print(model$coef)
# ar1        ma1
# 0.8019811 0.3693047
```

The information needed to recreate an ARIMA model in *R* will be saved as a Source as depicted in Listing 3.4. In order to be able to save all the information necessary for a model in one Source, we define an array like structure through roles of the Source. The first value represents a key, the second the index and the third the actual value. With this in mind the data in the Source from Listing 3.4 could be seen as two arrays and a singular integer value: `coef = {0.8019811, 0.3693047}`, `arma = {1, 1, 0, ↪ 0, 1, 1, 0}`, `var = 1.027177`.

Listing 3.4: Source representing the ARIMA Parameters as shown in Listing 3.3.

```
SOURCE SampleArimaModel TYPE LIST CONST
ROLES (key, index, value):
  ["coef", 1, 0.8019811],
  ["coef", 2, 0.3693047],
  ["var",  1, 1.027177],
  ["arma", 1, 1],
  ["arma", 2, 1],
  ["arma", 3, 0],
  ["arma", 4, 0],
  ["arma", 5, 1],
  ["arma", 6, 1],
  ["arma", 7, 0]
END
```

R-code to recreate the model is shown in Listing 3.5. Since the model used in *R*'s arima functions consists of a list of lists, first a list has to be declared. The variance (sigma2) and order (arma) can simply be added to the list. For the coefficient vector, firstly the values have to be defined and then named. The named vector is lastly added to the list, completing the model.

Listing 3.5: Recreating an ARIMA model in R.

```
model <- list()
model$sigma2 <- 1.027177
model$arma <- c(1,1,0,0,1,1,0)
coef <- c(0.8019811, 0.3693047)
names(coef) <- c('ar1', 'ma1')
model$coef <- coef
```

With a Source containing a Model and a Source containing the corresponding time series, a Check is needed to bring these two parts together. The Check is shown in Listing 3.6, since the calculation is covered by *R*, only the invocation of the arima() build-in function is necessary. The task of this build-in function is to recreate the given ARIMA model in *R* and then call *R*'s functions in order to determine the scores, which are then returned for evaluation during the execution of the Action.

Listing 3.6: DSL Check code for the ARIMA Check.

```
CHECK ArimaCheck ON LIST ts(time, value)
               WITH LIST model(key, index, value):
   RETURN LIST ARIMA(ts, model);
END
```

The main functionality of the *R* implementation is shown in Listing 3.7, where the function *arimacheck* is defined. It returns the scores of all data points, while also generating a plot showing the original time series (black line), the predicted time series (blue line), the threshold violations (red lines), as well as the scores (points)

with the threshold (black horizontal line) at the bottom. Sample plots are depicted in Figure 3.3 and 3.4, seen is the arima check applied to the introduced time series, once without anomalies and once with two injected outliers where the value was increased by 25 percent at index #51 and #83. In ARIMA models a predicted value depends on its predecessor, therefore an outlier in the time series, if not adjusted, will cause a few following predictions to be inaccurate, which can be seen at the zigzag pattern of the blue prediction line after an anomaly occurs. Therefore, the score of data points following the outlier most likely violates the threshold also. This can be seen in Figure 3.4, where the artificially introduced anomalies are found with three threshold violations each following the actual outlier. In order to not flood the analyst with too many error reports, side by side anomalies will be bundled together.

Listing 3.7: R-Code for the ARIMA Check.

```
arimacheck <- function(ts, model, threshold) {
  newModel <- Arima(ts, model = model)
  predictedts <- ts - newModel$residuals
  scores <- abs(newModel$residuals) / sqrt(model$sigma2)

  #plot
  plot(ts, lwd=2, type='l')
  lines(predictedts, col = 'blue', lwd = 2)
  points(scores)
  abline(0,0)
  abline(threshold,0)
  for(i in 1:length(ts)) {
    if(scores[i] > threshold) {
      abline(v=i, col = 'red')
    }
  }
  return (scores)
}
```

An Action, using the previous example Sources, can be seen in Listing 3.8. The time series Source used in this example is the one shown in Listing 3.2 with the two introduced outlier mentioned earlier at index #51 and #83. With a threshold of 3 the following error reports would be produced from the rule execution:

- Error at {51, 52, 53, 54}[2]
- Error at {83, 84, 85, 86}[2]

Listing 3.8: Action representing a sample ARIMA Check execution.

[2]The Report could also show the timestamps of the errors found, but in this example the time series is generated artificially and therefore the indeces are the timestamps.

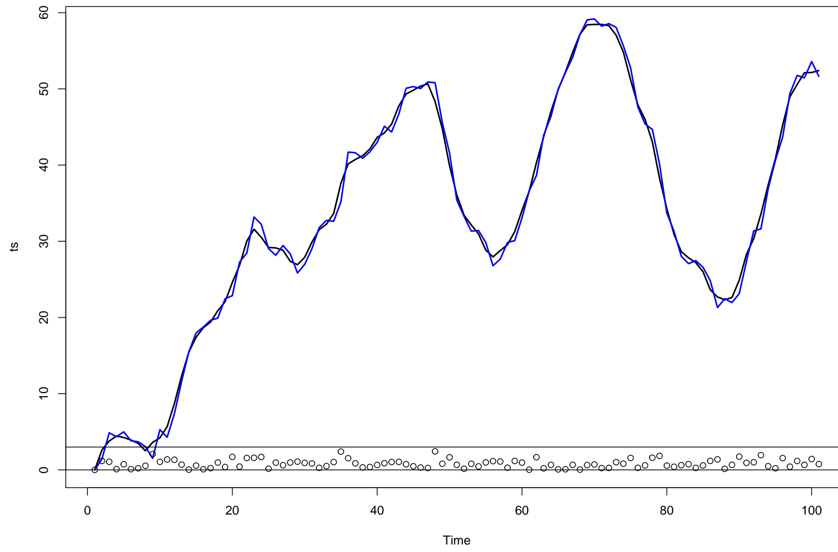

Figure 3.3.: Plot of the *R*-ARIMA Check with the sample time series.

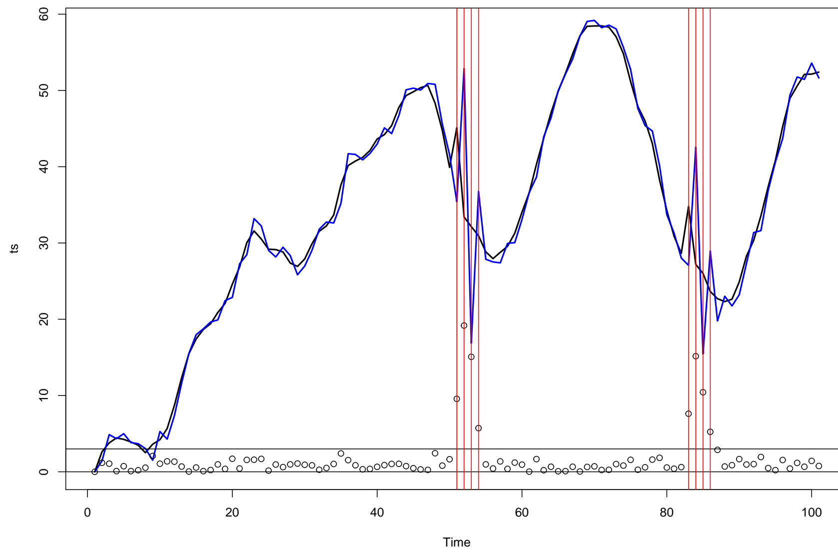

Figure 3.4.: Plot of the *R*-ARIMA Check with the sample time series and two introduced outlier (25% increase at 51 and 83).

```
ACTION SampleArima:
  EXECUTE ArimaCheck ON SampleTimeSeriesOutlier
  WITH SampleArimaModel
  RESULT IN ERROR ABOVE 3
END
```

3.2.2. Seasonal time series

Using the described approach, we can also describe and check seasonal time series using SARIMA models. A time series in SARIMA models (see e. g. [20]) is described by multiple components. They are driven by a series of random components e_t with zero mean and equal variance σ^2. The AR component is a linear combination of the previous values of the time series and the current random component. The MA component is a moving average of the previous random components. The "I" stands for "Integrated" meaning that the differenced series is looked at. The seasonal part references back to the values in the previous season, for example $S = 7$ for the same day in the previous week. The seasonal part can also be described using AR, MA or I components. Overall, the order of an SARIMA model is described by $(p, d, q) \times (P, D, Q)_S$, where the first triple defines the order of the AR, I, and MA components in the non-seasonal part, and the second triple defines these orders for the seasonal component.

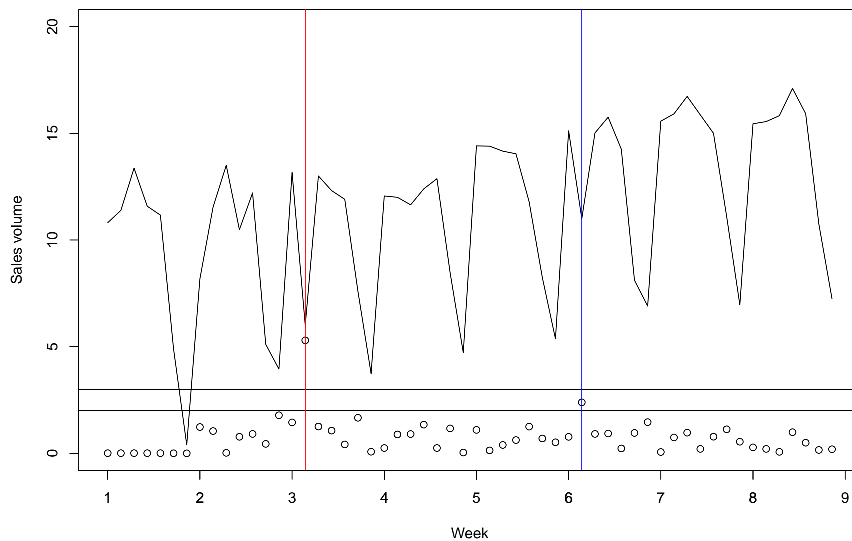

Figure 3.5.: Sales time series with daily ARIMA scores.

For example, the model we use for the time series in fig. 3.5 is an $(1, 0, 0) \times (2, 1, 0)_7$

Listing 3.9: Source for time series sales data.

```
SOURCE TimeSeriesSales TYPE LIST QUERY
                       ROLES(time_id: IDENTIFIER):
DATABASE ts NATIVE
  SELECT '' || week || '-' || day as time_id,
         sales_volume
  FROM time_series_sales
END
```

model, which means that we have an AR model that only uses the last value and an $S = 7$ weekly seasonal component that uses 2 AR components. Furthermore, the seasonal model is differentiated once as $D = 1$. We have no MA parts here. The equation for this model series is:

$$(1 - \mathrm{ar}_1 B)(1 - \mathrm{sar}_1 B^7 - \mathrm{sar}_2 B^{14})(1 - B^7) y_t = e_t \tag{3.8}$$

Here, B^l is the backshift operator, meaning $B^l y_t = y_{t-l}$. The parameter values are $\mathrm{ar}_1 = 0.6301$, $\mathrm{sar}_1 = -0.5609$, and $\mathrm{sar}_2 = -0.1503$. The variance of e_t is estimated to $\sigma^2 = 2.12$. Using this model we can calculate a forecast $\hat{y}_t$ by resolving (3.8) for y_t and setting $e_t := 0$. The forecast interval is calculated using the variance of e_t.

Our check now computes $\hat{y}_t$ for every time step t using the previous values. The absolute difference between the forecast and the actual value, normalized with σ, defines the score for each point:

$$\mathrm{score}_t = \frac{|y_t - \hat{y}_t|}{\sigma}$$

For the forecast calculation of later points, we replace outliers where the score is larger than 2 with the forecasted value to avoid that a single error influences the forthcoming forecasts and produces consecutive errors.

As the check code uses a built-in method `arima` that internally calls a suitable standard function of the R statistics package, the code is rather simple, cf. listing 3.6. Data to be checked is specified using a relational data source, see listing 3.9.

The model itself is described in RADAR using a constant Source, that is not computed by executing an external query, but rather uses a fixed structure throughout. This constant Source lists the order of the model, the coefficients and the variance of e_t. An example matching the above model is shown in listing 3.10. Since it is very difficult to find this model manually, especially for the DQ manager, we will use a profiling component described in section 4 to automatically suggest a model from analyzing historical data.

Finally, an action puts it all together. It calls the ARIMA check using the model and the time series data source, and assigns the appropriate roles, as shown in listing 3.11.

Continuing the example figure 3.5, we see that the sales volume data has a weekly seasonal pattern. The circles at the bottom denote the computed scores; the upper

Listing 3.10: Source for sales data ARIMA model.

```
SOURCE ArimaModelSales TYPE LIST CONST
                       ROLES (key, index, value):
   ["coef", 1, 0.5803],
   ["coef", 2, -0.4666],
   ["coef", 3, -0.1823],
   ["var", 1, 2.374],
   ["arma", 1, 1],
   ["arma", 2, 0],
   ["arma", 3, 2],
   ["arma", 4, 0],
   ["arma", 5, 7],
   ["arma", 6, 0],
   ["arma", 7, 1],
   ["coefnames", 1, "ar1"],
   ["coefnames", 2, "sar1"],
   ["coefnames", 3, "sar2"]
END
```

Listing 3.11: Action for sales time series data quality rule.

```
ACTION ArimaCheckSales:
  EXECUTE ArimaCheck
  ON TimeSeriesSales(time_id: time, sales_volume: value)
  WITH ArimaModelSales
  RESULT IN WARNING ABOVE 2.5 AND ERROR ABOVE 3.5
END
```

horizontal line indicates the error threshold of 3, the lower horizontal line indicates the warning threshold of 2. Note that we don't have scores for the first week as we need historic data for forecasting. At time 3-2 (second day of week 3), there is an error, the value of the sales volume is too small for a non-weekend day. Due to the seasonal model, this error can be detected. On the other hand, at time 6-2 there is a warning as the value is rather low but not as exceptional compared to 3-2. Yet, the data steward may know that these values are due to some shops having been closed for a public holiday. On the other hand, it might also be due to an error in data provisioning and in that case the error and the warning should be observed. Thus, days 3-2 and 6-2 will be reported as suspicious. It would be impossible to detect both types of such errors or warnings without using a seasonal time series, as both values are within the normal range of values.

Similar to these time series checks, we have implemented multidimensional quality checks. Syntactically, those are pretty similar to the time series checks, with the difference, that the Action provides the Check with information as to which attributes contain the cube metric/fact value, cube dimensions and cube time dimension, respectively. The Check itself is rather simple, basically invoking a built-in function computing a score for the passed data cube by comparing it against a desired cube model. This model is specified as a constant data source similar to the one presented in listing 3.10 for time series, just with different parameters. Based on the computed score the Action can issue warnings or errors, as before.

Since the constant data sources needed to provide the models for the advanced quality checks tend to be lengthy and difficult to discover, it is important to assist the DQ manager in specifying such sources. This is done by our profiling component as will be explained below.

3.3. Multidimensional Data

In this section, we describe our approach to quality checks for multidimensional data. By multidimensional data, we mean Cube data, i.e. facts that are described by dimensional attributes, typically including a time dimension, and metrics values. Metric values can be aggregated over one or more dimensions. Dimension values might be grouped to form hierarchies.

As multidimensional data is ubiquitous in Data Warehouse environments, we pay special attention to this kind of data. Treating multidimensional data as normal data using the previously described checks is possible, however it might miss to find more subtile quality problems associated with specific dimensions.

As an example, assume sales data grouped by the dimensions like product, customer, region, sales agent, promotion, etc. Furthermore assume that some error caused that an old definition of the product catalog has been used during data load. This means that all data from this load is assigned to wrong dimensional values for the product dimension, while all other dimensions are properly encoded. Thus the overall sum of the revenue is correct, however specific aggregations that include the product dimension will show wrong values. Thus we need quality checks that pay special attention

to individual aggregate values. The basic idea used here is described in [17] in the context of network data.

This section is organized as follows. First, we describe the DSL perspective and extensions to describe multidimensional data and multidimensional checks. In the next subsection, we describe the profiling process that generates multidimensional checks. Finally, we describe the method used to run the checks and to minimize the number of generated error messages.

3.3.1. DSL extensions

We assume to have the cube data in a relational form, i.e. as a list of records. A multi-dimensional source is described using standard DSL syntax. The specific dimensional structure is described using appropriate roles:

Listing 3.12: DSL code for a sales cube Source.

```
SOURCE SalesCube TYPE LIST QUERY:
DATABASE dwh NATIVE
  SELECT dd.day, dd.month,
         c.product_id, c.product_group_id,
         c.customer_id, c.region_id,
         c.sales_volume
  FROM sales_cube c
  JOIN date_dim dd ON dd.day_id = c.day_id
END
```

A cube check is complex, thus we hide it within a build-in function:

Listing 3.13: DSL code for a sales cube Source.

```
CHECK CubeCheck ON LIST cube(time, dim, metric)
                   WITH LIST cubemodel:
  RETURN LIST CUBESCORE(cube, cubemodel);
END
```

The cube checks needs to know the metadata for the source, i.e. the dimensions, metrics and the time dimension. We use the role concept of the DSL to define this metadata. An Action will look as follows:

Listing 3.14: DSL code for a sales cube check Action.

```
ACTION SalesCubeCheckDay:
  EXECUTE CubeCheck ON SalesCube(day: time,
                                 product_id: dim,
                                 product_group_id: dim,
                                 customer_id: dim,
                                 region_id: dim,
                                 sales_volume: metric)
```

```
  WITH SalesCubeModelDay
  RESULT IN WARNING ABOVE 2 AND ERROR ABOVE 3
END
```

The cube model is stored as a constant (alternative: as a database table). It looks as follows. For each cell where a model exists, the cell is identified by the dimensional attributes (this list must match the dimensional attributes of the cube). For each cell, we define the model as a normal distribution using a mean μ and a standard deviation $\sigma = \sqrt{\sigma^2}$:

Listing 3.15: DSL code for a sales cube model source.

```
SOURCE SalesCubeModelDay TYPE LIST CONST
ROLES (dim, dim, dim, dim, mean, sd):
  [17, 28, 13, 104, 100, 30],
  [26, 28, NULL, NULL, 17000, 400],
  [NULL, NULL, NULL, NULL, 100000, 3000]
END
```

In this examples, three models for difference cube cells are defined. The first line is a very specific model for a base cell of the cube. The second line is a model that applies to product 26 within group 28. The third line defines that the overall cube sum per time unit has a mean of 100000 with a variance of 3000.

For other time units, we need different models and a new action:

Listing 3.16: DSL code for a sales cube check Action.

```
ACTION SalesCubeCheckMonth:
  EXECUTE CubeCheck ON SalesCube(month: time,
                                 product_id: dim,
                                 product_group_id: dim,
                                 customer_id: dim,
                                 region_id: dim,
                                 sales_volume: metric)
  WITH SalesCubeModelMonth
  RESULT IN WARNING ABOVE 2 AND ERROR ABOVE 3
END
```

Listing 3.17: DSL code for a sales cube model source.

```
SOURCE SalesCubeModelMonth TYPE LIST CONST
ROLES (dim, dim, dim, dim, mean, sd):
  [10, 89, NULL, 102, 3000, 30],
  [23, 42, NULL, NULL, 170000, 400],
  [NULL, NULL, NULL, NULL, 3000000, 3000]
END
```

3.3.2. Profiling

The profiling for multidimensional data builds a model using historic cube data. For the prototype, we assume that the historic data is error free. Later, it will be useful to identify outliers in the historic data prior to model building in order to improve model quality.

The main challenge for this profiling module is to select a useful subset of cells. It is not feasible to build a model for every cell as there are too many cells (exponentially many in the number of dimensions). Furthermore, there are cells with very few data. Building a model using only few data records will generate spurious models and thus far too many false positives during later operation.

Thus our criteria for selecting cells will be the amount of data available in the cells. We measure the amount using the number of records in the fact table, no matter how large the metric value in every record is.

We build the iceberg cube, i.e. the cube consisting of all cells that fulfill a certain threshold condition. The threshold is given as a parameter T. We only build a model for cells where more than T data records are available. For each of these cells, we group the data per time unit and then calculate the estimated mean and standard deviation.

For efficient computation of the iceberg cube, we use the FTL algorithm described in [19].

3.3.3. Running the check

Running the check essentially means to compute all cube cells for which a model is available and then to calculate the Z-score for every time step of every cube cell. The Z-score is returned. The action can evaluate the score against the warning and error boundaries.

Alter Text:

Listing 3.18: Syntax for list methods.

```
lstmethod ::=
  ARIMA '(' lstexpr ',' lstexpr ')' |
  CUBESCORE '(' lstexpr ',' lstexpr ')'
```

ARIMA implements an ARIMA time series check and returns one score for each time step of the series. The first parameter of ARIMA is the time series, it is expected to be a list with two roles: TIME and VALUE. The TIME role is used for identification purpose only and not further interpreted. The records are expected to be ordered by time and to have no missing values. The second parameter is the ARIMA model. Each record in this list must have three roles: KEY, INDEX, and VALUE. An example is given as follows:

Listing 3.19: Example for the model parameter.

```
SOURCE ArimaModelSales TYPE LIST CONST
```

```
                        ROLES (key, index, value):
   ["coef", 1, 0.5803],
   ["coef", 2, -0.4666],
   ["coef", 3, -0.1823],
   ["var", 1, 2.374],
   ["arma", 1, 1],
   ["arma", 2, 0],
   ["arma", 3, 2],
   ["arma", 4, 0],
   ["arma", 5, 7],
   ["arma", 6, 0],
   ["arma", 7, 1],
   ["coefnames", 1, "ar1"],
   ["coefnames", 2, "sar1"],
   ["coefnames", 3, "sar2"]
END
```

The return value of the call is a list containing a record for each time step with the roles IDENTIFIER and SCORE. This list can directly be returned from a check.

4. Profiling

4.1. Introduction

Data profiling is used to gain information about existing data, the idea is to get a thorough understanding of what the data looks like. This projects aim is to use data profiling techniques to generate suggestions for data quality rules, meaning rules describing data as it should be. The profiling component generates rules using the *RADAR* DSL. This chapter will give a brief overview over the concepts and design decisions of the data profiling component.

In order to ensure data quality, a huge set of rules is necessary to regularly test as much quality aspects and parts of the data set as possible. When writing rules, two challenges arise. First, the effort to write huge numbers is high and thus a good coverage of the data with rules might not be achieved. Second, finding good parameters and thresholds for complex rule types by hand is extremely difficult. This holds true especially for rules that define a statistical model for the data and check the conformance of the current data with the model (see chapter 3). To simplify these tasks, we developed a profiling component that examines existing data and suggests rules and parameters for these rules. The data quality steward can then inspect these suggestions and decide, using additional business knowledge, which candidates should be made active. Manual inspection is important, as profiling tools can only detect rules that are currently fulfilled within the data, without knowing whether they hold in general.

We follow a modular approach and provide various profiling modules. Each module is focused on one type of profiling and outputs specific types of rules. As a simple example, the NOT NULL profiling module examines a column of data and counts the current number of NULL and NOT NULL entries in this column. Based on these statistics, the module suggests either a hard NOT NULL rule or a soft NOT NULL rule, stating that not more than a certain fraction of the values may be NULL, or both. For the soft rule, we need to find reasonable parameter values. Another example is a profiling module for a time series rule, that has to estimate the parameters of the time series model.

The UI to steer the profiling is two-fold. First, we have a basic profiling UI that targets the generation of huge amounts of basic rules. This UI shows the data sources and their attributes in a grid-like view and automatically suggests basic profiling modules that match the data type of the attributes, i. e. numeric range profiling for numeric columns, NULL profiling for every column, etc. The goal is to quickly generate rules to check the individual column values with a good coverage. The profiling functionality here is similar to commercial data profiling tools in principle, but with the extension,

Listing 4.1: "Profiling output"

```
The following rule holds with 0 error(s) and 0 warning(s):
  EXECUTE NotNullCheckProb
  ON budget(BUDGET) WITH 0.03
  RESULT IN WARNING ABOVE 0.95 AND ERROR ABOVE 0.99
The following rule holds with 1 error(s) and 0 warning(s):
  EXECUTE NotNullCheck
  ON EACH budget(BUDGET)
  RESULT IN ERROR
Errors: [movie = 48417]
```

that quality rule candidates are automatically generated based on the profiling results. The other UI, called advanced profiling, targets more complicated rules where the profiling needs more information to run. An example is the time series profiling, which needs to know which attribute of a data source contains the time id and which attribute contains the value of the time series. Here, running automatically over every combination would produce extreme load and generate lots of meaningless rules. Thus we assume the data steward can steer the process better using business knowledge of the data. For the basic profiling UI, we use one generic kind of dialog. For the advanced profiling UI, each profiling module defines its own UI components and parameters that are needed to steer this module.

4.2. Profiling, Statistics and Rule Generation

In this section, we describe the basic idea behind our profiling mechanism. Traditionally, profiling means to generate some statistics about data. Basic statistics for a single column include the percentage of NULL values, the number of different values occurring, and the percentile values. In our profiling module, these values are used to generate profiling rules. Each profiling module uses the data or some statistics generated from the data in order to build candidate rules to describe the data. Each profiling module can generate one specific kind of rule or one set of rule types. A profiling module for NOT NULL rules can e.g. generate a hard NOT NULL rule that requires each instance of a column to be NOT NULL, or a percentage NOT NULL rule that requires a certain percentage of instances to be NOT NULL. Suppose for instance that the NOT NULL statistic reveals that 97 percent of the instances of a column are NOT NULL. This could lead to the generation of two rule candidates: one hard rule that requires each instance to be NOT NULL and one probabilistic NOT NULL rule that assume the probability of a value to be NULL is 0.03. In general, the idea is that at least one rule candidate describes the data as is exactly, and possibly multiple alternative candidates describe the data including some potential errors. Thus, the result of the above profiling could be presented to the user as shown in listing 4.1.

4.3. Profiling Methods

In the following, we present details of selected profiling modules in order two better show our approach and the way of thinking behind these modules. In general, every module might either output rules that are completely satisfied by the current data or rules that already declare some data as errors or warnings. For these rules, every module has a `maxoutlier` configuration parameter that steers how many errors are acceptable for a rule to ultimately be suggested.

4.3.1. NOT NULL profiling

In the rule catalog, we have two kinds of NOT NULL rules. The first is a hard rule that outputs an error when there is at least one NULL entry. The second is a soft variant that is based on statistical assumptions. The assumption is that each value has a certain probability p of being NULL, independently of the other values. Thus the random variable X that counts the number of NULL values follows a Binomial distribution with the parameters p and n, the current size of the data source. The score that the rule outputs is the probability that X is smaller than the current value, i. e. $P(X < x)$, where x is the current number of NULL values. Typical thresholds are then 0.95 for warnings and 0.99 for errors, meaning that the probability of having an as-large or larger number of NULL values is smaller than 0.05, as $P(X \geq x) = 1 - P(X < x)$ (under the chosen assumptions). Thus the soft NULL rule essentially performs a one-sided Binomial test, and the thresholds determine the significance level. However, we use 0.95 and 0.99 and $P(X < x)$ instead of 0.05 and 0.01 and $P(X \geq x)$ as our system uses the general convention that larger scores indicate more severe quality problems.

The NULL profiling module starts by counting the current number x of NULL values in the chosen attribute, and the overall size of the relation n. The module suggests a hard rule in case that the overall number of NULL values is smaller than the “max errors” parameter. A soft rule is suggested in case that at least one NULL value exists, using the estimate x/n for the parameter p. However, when this estimate for p is larger than 0.5, we don’t suggest any rule, as in this case NULL values are more probable than NOT NULL values. The standard thresholds that are suggested are 0.95 and 0.99 for warnings and errors, respectively.

As an example, assume we have a column with 1000 entries where only 2 are NULL. In this case, it is possible that there is a business reason for these NULL values, or these values might be data quality problems. Accordingly, the profiling module will suggest a hard NOT NULL rule and output that this rule will generate two errors based on the current status, and it will additionally suggest a soft NOT NULL rule with $p = 0.002$. The steward then must make a decision between these two rules based on his domain knowledge. In case that a column does not have any NULL values, only the hard NOT NULL rule is suggested.

Listing 4.2: DSL code for probabilistic NOT NULL check.

```
CHECK NotNullCheckProb ON LIST lst(value) WITH p:
```

```
    LIST cntNull := SELECT COUNT(*) FROM lst WHERE value IS
        ↪  NULL;
    LIST cntAll := SELECT COUNT(*) FROM lst;
    score := BINOM(cntNull-1,cntAll,p);
    RETURN score;
END
```

4.3.2. Range profiling

A range rule states that the value x of a column must be between two values l and u. The score is 0 when $l \leq x \leq u$, and it is the absolute deviation from the lower or upper value in case that it is smaller than l or larger than u. The profiling module for this rule examines the current column values. The first rule that is output is a rule that uses the minimum and maximum value occurring as l and u, respectively. This rule does not return any error for the current data set. The second rule copes with outliers. For this, the lower and upper quantile ($Q1$ and $Q3$) of the values are computed and the typical outlier definition using the interquartile distance $IQD = Q3 - Q1$ is applied. Thus we define $l = Q1 - 1.5 * IQD$ and $u = Q3 + 1.5 * IQD$. Using these boundaries, we count the number of outliers and check whether the number is not larger than "max outliers". When this is true, we suggest the alternative range rule.

4.3.3. Foreign key dependencies

Foreign key profiling works essentially exactly as NULL profiling, because we use the current number of violations of the foreign key as basic statistic. Based upon this statistic, the module chooses to suggest a hard and/or a soft rule to check the foreign key. However, foreign key profiling is an advanced method, because it would be too inefficient to try every possible attribute combination between any data sources. Thus, we expect the data steward to steer the profiling by defining foreign key candidates that should be checked.

4.3.4. Time series profiling

This module uses the R statistics package to estimate a time series model based on the data. For this, we need a data source with columns that carry the time identifier and the time series value. The whole time series is read and sent to R in order to retrieve an SARIMA model. The SARIMA model is converted into a constant data source (cf. listing 3.4) and will be stored in the rule repository along with the rule in case the rule is accepted by the data steward.

Based upon the SARIMA model, the time series is checked as described in section 3.2. The value of every data point is compared to the 95% and 99% prediction interval. In case it is outside either interval, the point is declared to be a warning or an error, respectively. The warnings and errors are counted and displayed to help the steward to decide whether to accept or decline the suggestions. In the future, we also

plan to integrate a graphical display of the time series, its model, and the detected warnings and errors in the UI component.

4.3.5. Cube profiling

A multidimensional data cube is a set of aggregate cells over a base relation that has dimensional attributes and metric attributes [2]. Almost any cube has a time dimension. To check the data of a cube, we profile how the data in the cells varies over time using the time dimension, and define a cube model based upon these statistics. Thus, a cube model is essentially a set of cell models.

During profiling, we need to choose the subset of cells for which models are computed, as the overall number of all possible cells is typically extremely large (growing exponentially in the number of dimensions). Our main criteria is the availability of enough data for a cell to be included in the model. First, the user chooses a time interval for the profiling (i. e. daily, weekly, monthly). Then the profiling process calculates a cube for every interval separately. In order to limit the computational power and memory requirements, we limit these cubes to Iceberg cubes [31] using a threshold parameter. This means that only cells where the metric value is above the threshold are included in the cube. Then, we run through all time intervals and collect statistics about all cells that exist in every time interval. This ensures a focus on the higher-level aggregate cells that contain enough information for reliable models.

Currently, we use a Gaussian distribution to model each cell's value. Thus we use the sequence of metric values computed for the current cell for every time interval to estimate μ and σ. The overall cube model consists of all these values for the selected cells. The cube model is provided in the format of a constant data source of our quality rule language (similar to time series above) and can thus be easily used in quality checks, if the corresponding suggested rules are accepted by the data stewart. The final steps are also identical to the time series profiling: we use the model to calculate the current number of warnings and errors and suggest the rule together with this information.

4.4. Design

In the UI, we distinguish between *standard* and *advanced* profiling. The differences are, that the standard UI will output many suggestions for e. g. different columns of a table for multiple kinds of simple data quality rules, whereas the advanced UI one only generates suggestions for one kind of rule per profiling. Even though the output of both methods varies in quantity, the form is the same; one or more rule candidates written in the DSL with additional information, e.g. about potential outliers. Therefore, we decided to split these methods only UI-wise and provide a common interface in the back-end. An example for *standard rule suggestion* would be the execution of the multiple profiling methods over multiple data sources, this could be triggering a null and unique profiling, and, depending on the data type, range or string length profiling for each column of a given table. Triggering the profiling for a time series model,

an ARIMA model for instance, on a given time series (e.g. the daily revenue of a business over the last two years), is an example for an *advanced rule suggestion*. This architecture enables an easy way to increase the amount of provided rule generation methods, due to the fact that the profiling service only has to provide one interface, regardless of how the UI works.

Internally the Profiling-Service contains a number of profiling methods, one for each supported Check to generate Actions for. In order to not have to process the same source data every time a profiling method is triggered, it is possible to group profiling methods, which use the same Source, together, to calculate all statistics about them in the fewest possible iterations over the data (modules might need source data in a specific order, which can result in reading the same source data multiple times).

4.5. Architecure

The architecture, in form of a class diagram, is shown in figure 4.1. The profiling service is mostly defined by its *ProfilingMethods*, which will calculate statistics about source data and use those to generate rule candidates. Each profiling method will generate rules (Action DSL code) for a specific Check, as indicated by its name (e.g. NullCheckProfiling). In the shown diagram only three different methods are shown, this is meant as an example of how profiling methods may look like.

The ProfilingManager supervises and handles all executions of profiling methods. It contains references to all current queued methods and the source data they need. Further, the manager requests the source data from the DBManager and distributes it to the dependent profiling methods. The first step is to execute each method unoptimized, meaning every Source is read for each ProfilingMethod instance. In order to increase the performance, an execution plan has to be generated which orders database accesses in such a fashion, that data has to be read the fewest times possible. Dependencies of data sources have to be minded. A method may need one Source before it can handle another one, which may result in the need to read Sources multiple times.

4.5.1. ProfilingMethod

The ProfilingMethod is an abstract class serving as a template for all different kinds of profiling methods. Each method corresponds with a Check and generates Actions for that Check. The class contains a map of profiling parameters, which can be used to adjust the profiling. Mandatory methods for all profiling methods are nextValue() and getResult(). While the latter one is used to trigger the final result calculation, the nextValue() method is used to sequentially feed data to a method. This was introduced to enable multiple profiling methods access to the same source data while reading it only once. A manager class reads all source data and calls all profiling methods nextValue() methods, which need that data. Each profiling call, which consists of (at least) profiling type and source data, instantiates a new ProfilingMethod object, which is registered at the ProfilingManager.

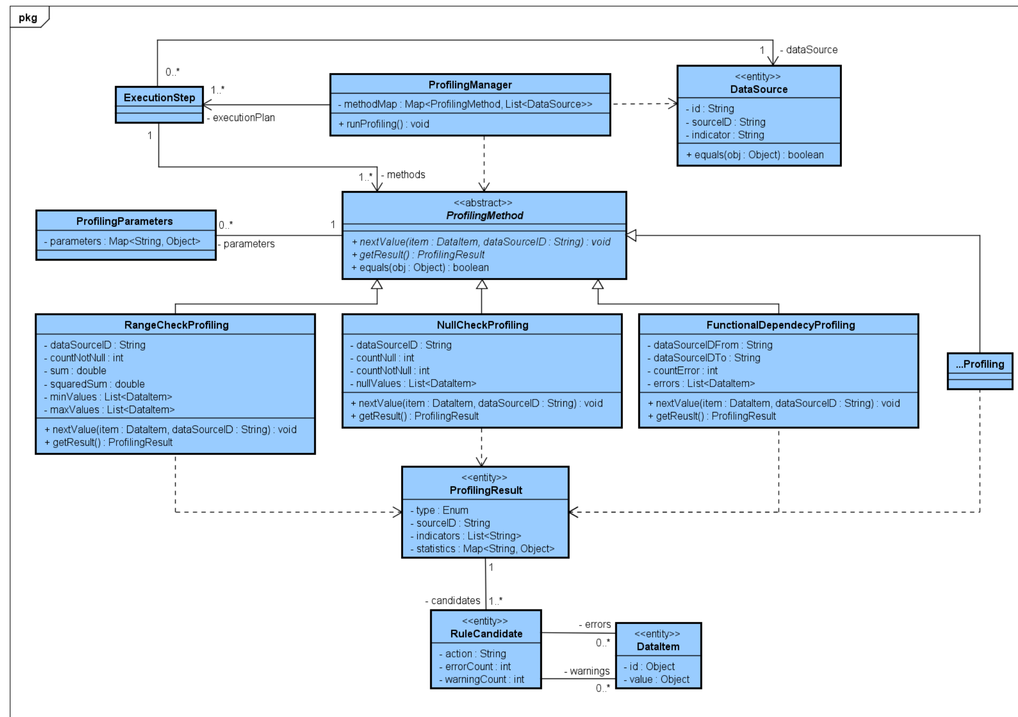

Figure 4.1.: General architecture of the Profiling-Service.

4.5.2. ProfilingResult

The information gained from a ProfilingMethod is saved in the ProfilingResult entity. It contains information about the execution, like which method was used and on what source data. Also saved are key statistics about the source data, calculated by the method and used to generate the rule candidates. The candidates are the last and most important part of the ProfilingResults. One result may contain multiple rule candidates, one of which represents the data as is, meaning that outliers were not excluded from the calculations. Other rule candidates may represent the data with varying methods of outlier removal applied. A rule candidate consists of an Action and two lists of tuples. The lists contain data which either produce an error or a warning, respectively, if the rule was enforced on the current source data. This is a result of the outlier consideration of the profiling module. The idea behind outlier removal in profiling for rule candidates is, that the source data might already have some data quality issues.

4.5.3. API

Information needed for the profiling component is foremost a Source, an indicator for data (in relational terms a column) and the desired profiling method. Alongside the selected method optional parameters might be added. Listing 4.3 shows a sample REST call to trigger a profiling for a RangeCheck and a NullCheck on the Customers age and also a FunctionalDependencyCheck for the Customers City and ZIP.

Listing 4.3: Sample REST call of for the Profiling-Service.

```
HTTP/POST profiling-service/profile
[{
"method": "range",
    "data": {
    "sourceID": "Customer",
    "indicator": "age"
    },
    "params": {
        "maxOutlier": 10
    }
},
{
    "method": "null",
    "data": {
        "sourceID":"Customer",
        "indicator": "age"
    }
},
{
    "method": "functionalDep",
```

```
    "from": {
        "sourceID":"Customer",
        "indicator": "City"
    },
    "to": {
        "sourceID": "Customer",
        "indicator": "ZIP"
    }
}
}]
```

5. Feedback Loop

5.1. Return values from checks

To structure the collection of feedback, the return values from checks must be standardized. This is described in section 2.2.7 of this document. The results of the checks are stored in the reports databse.

In the next step, feedback is given by the user through the GUI. For each individual record returned by a rule (errors and warnings) the following classification is needed and should be done by the user:

- **A: error**. It is a real data quality problem. The issue should eventually be fixed in the underlying database.
- **B: normal value**. No error, the value is correct.
- **C: outlier**. No error, but an unusual value. In this case, the rule is ok, however the values should later on be ignored.

This feedback indicates changes to the underlying rule as listed in figure 5.1.

	Warning	Error
A	stronger rule: report as error	no change
B	weaker rule: don't report	weaker rule: don't report
C	no change; ignore only this issue	no change; ignore only this issue

Figure 5.1.: Rule changes indicated by user feedback.

Please note that feedback in general might be contradictory. I.e. a user might confirm an age of 101 for customer X while marking the same age for customer Y as an error. In this case, a range profiling method could not be written that takes both types of feedback into account. In this case, setting the age 101 of X as an outlier value would have been the better option.

After collecting feedback, the profiling should have a new entry page where all actions are listed that currently have unprocessed feedback. The actions could by sorted in decreasing order of the number of feedbacks stored for each action.

For each of these actions, a re-profiling can be started, that tries to update the action in order to make it a better fit with the collected feedback. Some re-profiling methods also generate multiple suggestions, and some also could suggest to delete the rule.

5.2. Rule types

We distinguish several rules types according to the following list:

- Hard rules: these output only errors, but no warnings. Feedback can be A, B or C based on individual records or values (Type 1 or type 2 feedback). Examples: NOT NULL, Range, Functional Dependency, Unique, Referential Integrity
- Statistical rules: these output a score, and classify larger scores into warnings or errors. This type has two sub-types:
  - One score for each individual data item or smaller groups of data items: Time series, Cubes. There can be a single feedback for each data item / group of type A, B, and C.
  - Only a single score for the whole data set: PercentageNull, Distribution. Here we only have a single feedback record. Only type A or B are useful. Type C would indicate to delete the rule. Perhaps the GUI should not allow to assign C to these records.

Note: For the last type, it would be interesting to assign individual feedback to the underlying data items. However, this is out of the scope of our prototype.

5.3. Re-profiling for individual actions

The exact algorithm for re-profiling depends on the type of check underlying the action. Re-profiling is only possible if there is a re-profiling method implemented for the target check. In general, results of type C are ignored during the re-profiling, as they are declared to be outliers. Thus the input to the profiling is a set of records / values, that are classified as A (error) or B (correct value) or D (values that have no feedback so far).

The number of records in these groups that are feed into the re-profiling is denoted with $|A|, |B|$, and $|D|$. The whole number of records is denoted n. In the case of re-profiling, n does not include type C records, as they are ignored.

5.3.1. Re-profling for NOT NULL

The following suggestions are generated:

- Keep the current NOT NULL rule. This implies that all B records (NULL values declared as normal) are converted to C records (outliers).
- When there are any B or D values, then a percentage NULL rules is generated. p is computed as: $|B|/n$. This means that there will probably still a warning or an error as long as there are type D records (no feedback) as these are not used to increase p appropriately.

5.3.2. Re-profiling for RangeCheck

The range check generates a single suggestion. The goal is to enlarge the admissible range to include as many B values as possible while not including any A values into the range. Thus we increase the upper bound to the largest B values that is still smaller than the smallest A value above the current upper limit u:

$$u' = \max(u, \{b \in B; b > u \wedge \forall(a \in A \text{ s.t. } a > u) : b < a\}) \tag{5.1}$$

A similar logic holds for the lower bound l:

$$l' = \min(l, \{b \in B; b < l \wedge \forall(a \in A \text{ s.t. } a < l) : b > a\}) \tag{5.2}$$

A small example: we have a range of 0 to 100, and reports about the values 101, 102, 103, and 104. The feedback ist B, A, B, C. Thus we first increase the upper bound to 103 to make 101 and 103 valid values. However, now 102 is no more reported. Thus we decrease the upper bound to 101 to make it an error again. Finally, we change 103 to type C (outlier).

An alternative would be to re-label A to B and to set the upper bound to 103. However re-labeling A to B is rather dangerous and thus discouraged.

5.3.3. Re-profiling for percentage NOT NULL

Here, we only have a single feedback. The report might be a warning or an error. We generate the following suggestions:

- Error, Type A: No change to the rule.
- Warning, Type A: Lower the error bound to the current score, so that the same score will be reported as an error later on.
- Warning or Error, Type B: Re-calculate p based on the current data so that the current distribution will no more be reported as an error.
- Warning or Error, Type C: Distribution is an outlier. Suggest to delete the rule. (or avoid feedback C for this type of rule)

5.3.4. Re-profiling for time series

First, we try to build new warning and error boundaries. The goal is to have all type A records as errors. Thus the first suggestion is to change the error boundary to the smallest score of any type A record. This implies to change all type B records that are above this boundary to type C records. Finally, the warning boundary can be increased up to the largest type B record that is currently a warning, making these records normal.

Second, we compute a new model. For this, we re-compute the time series, including only type B values. Type A and type C values are replaced with appropriate mean values. The resulting model is then applied using standard boundaries.

Overall, this yields two different rule suggestions that will be presented to the user. For each suggestion, we count how many B records will be warnings or errors (thus changed to C records) and how many type A records are only warnings or normal values (this can only happen for the second suggestion. The second suggestion also includes the possible to detect new warnings or errors that were normal before.

5.3.5. Re-profiling for cubes

Implementation similar to time series re-profiling.

6. Related Work

6.1. Existing solutions for data quality monitoring

The predecessor of this project, the Data Checking Engine (DCE) [18], had also been developed in our group. It is an SQL-based system for complex data quality rules. The DCE provides an interface for data quality managers to compose data quality rules in form of SQL statements. Even though templates were introduced to reduce the complexity of rule composition, especially regarding the creation of multiple rules of the same type for a lot of different data, the complexity of SQL and the underlying database schema still has to be known by the domain expert. Also, once written, rules were hard to read and to fully comprehend. Therefore, this project, as a successor, and especially the DSL copes with the mentioned problems the DCE has, in form of the separation of concerns via the split into Sources, Checks and Actions.

Endler et al. developed an architecture for continuous data quality monitoring in medical centers [12] which has many similarities with our proposal. They also handle data quality monitoring with a rule-based approach. In contrast to our solution, they did not introduce a DSL for describing rules. The most basic shown rules are simple boolean operations directly defined for the underlying database. That also shows the second part where our approaches differ, the rule system in [12] is designed for a special medical unified database system without the possibility to easily extend the database environment or adapt other schemas.

Recently, [27] presented an approach with a very similar goal to our work, namely automating data quality checks in large data sets. Whereas the goal and some of the features are similar to our approach (e. g. presence of a wide range of basic types of checks, methods to automatically suggest potential quality rules, integrating heterogeneous data sources), significant differences are also present. They use a more sophisticated optimization procedure to efficiently process the checks based on Spark and address differential quality checking in detail (which is not applicable to all checks in our system). On the other hand we provide more advanced data quality checks such as ARIMA time series model based checks or dimensional checks, require less programming knowledge by the DQ manager by separating concerns in the rule definition, have a comprehensive profiling component that can also detect complex quality rule candidates and facilitate what Jack Olsen calls "value rules" [24], which are checks that output a score instead of a clear correct/error decision.. To this end both approaches complement each other. Another approach for a rule based data validation system called GuardianIQ is introduced by David Loshin [23]. His concept is a quality monitoring based on business rules and executed in SQL. Focus of this solution also lies on the distinction between data cleansing and data quality in conjunction with

metrics to quantify data quality. However, we were not able to perform a closer comparison, since only rough information was given in a short paper from 2002 without newer publications.

Lastly, Batini et al. suggested a methodology for data quality assessment and improvement called Heterogenous Data Quality Methodology (HDQM) [5]. As the name suggests, their idea is a methodology applied to the processes of businesses. It provides guidelines - in form of a so called meta-model - on how to design the data quality process integrated in the target system. Whereas their system integrates in the business process requiring a complete remodeling of the users system, our approach can be seen as a complementary addition to apply to existing systems.

All in all, partial solutions similar to our approach exist, especially when looking at the DCE or the rule based approach from Endler et al. What is missing is a unified framework to cover all needs for data quality monitoring including most techniques for data quality checking and the ability to easily extend them, together with - especially - the aspect of monitoring heterogeneous data sources, which has neither been covered by the DCE, nor Endler's solution. All in all, the extension of compatibility for heterogeneous data sets based on a well-defined internal data model, the split between technical and subject-specific knowledge and the advanced types of quality checks such as ARIMA set our approach apart from existing solutions.

Database constraints are well-known, and reaseach about extended kinds of constrains like conditional functional dependencies (CFD) [7] is going on. *RADAR* is designed to capture all of these types of constraints. Furthermore, we also allow For these checks, all kinds of outlier detection [3] algorithms are interesting. Both, unsupervised algorithms can be implemented, as well as supervised algorithms that use some kind of model. Here, we are specifically interested in time series outlier detection [16] and outlier detection in multidimensional data (see e.g. [22]), as this fits perfectly with typical data warehouse data.

6.2. Individual implementation aspects

From the operational and query side, our work touches mutliple lines of work in the database research that we are going to mention here. First of all, our *RADAR*language allows to formulate queries integrated into an imperative language. This is similar on the one hand to approaches like Oracle's PL/SQL [14] or other stored procedure languages, and on the other hand to LINQ-approaches (language integrated queries), see [10]. Yet another approach is to integrate SQL with functional programming [6]. However, our combination is specific to the domain of data quality and offers specific features for that.

Furthermore, *RADAR* can act as a mediator [15] allowing to query and integrate multiple sources using a single piece of code. Thus the query optimization issues are quite similar to those in mediator-based systems. However, we don't define a global schema but allow the checks to directly access the individual sources, which is more appropriate for data quality checks. Similarly, a lot of work has been published in federated query processing. Those results are of interest for the further optimization

of the execution of our quality checks. This is still ongoing because in the initial version the processing is not performed in a federated way (with exceptions as explained in section 2.5).

RADAR can access both relational and NoSQL databases from the same code, which again is a generic problem in heterogeneous environments. Here, both query languages (i.e. UnQL, [9]) and systems have been developed. There are multiple mediator approaches, e.g. [4, 21, 32, 26] that allow to query relational and NoSQL-data from a unified SQL-like language. They are to some degree comparable, however they lack the parametrizing features vital for our approach. Fitting queries to document-based and relational data in a concise syntax with understandable semantics is still an issue.

Parametrizing SQL in the way we do it – parameterizing also schema elements like attributes and relations and with the option to e.g. use a compound foreign key as actual parameter for a single attribute parameter – is, as far as we know, a new feature. From our point of view it is vital to achieve the flexible modularization and reuse-capabilities we are targeting. The only comparable approach uses the Maude system [28] to rewrite parametrized SQL statements. However, we integrated the parametrization capabilities into our core language.

6.3. Advanced rules and profiling

An overview of data profiling can be found in [1]. There is a wide range of research activities about various constraint types and efficient methods to detect them. As an example, there has been research about extensions to FD in order to improve them. One result are conditional functional dependencies (CFD) as described in [7] and even an extension to them called eCFD presented in [8]. Techniques for discovering CFDs are shown in [13, 11], which is useful for our rule detection engine. All kinds of outlier detection [3] algorithms are interesting for our work as well. Both, unsupervised algorithms can be implemented and provided as rule types, as well as supervised algorithms that use some kind of model. Here, we are specifically interested in time series outlier detection [16] and outlier detection in multidimensional data (see e.g. [22]), as this fits perfect with typical data warehouse data.

6.3.1. Alte version

Data quality is a topic with many facets, a lot of different kinds of data quality exist; section 1.1 gave more insight on that. After defining data quality itself one can think about methods to check for data quality. Beside simple well established statistics of data, like row count, null count and unique count among others, the usual database constraints may apply as well, functional dependencies (FD) for example. There has been research about extensions to FD in order to improve them. One result are conditional functional dependencies (CFD) as described in [7] and even an extension to them called eCFD presented in [8]. Techniques for discovering CFDs are shown in [13, 11], which is useful for our rule detection engine. All kinds of outlier detection [3] algorithms are interesting for our work as well. Both, unsupervised algorithms can

be implemented and provided as rule types (specific functions in checks), as well as supervised algorithms that use some kind of model. Here, we are specifically interested in time series outlier detection [16] and outlier detection in multidimensional data (see e.g. [22]), as this fits perfect with typical data warehouse data. Our plan is to gradually support more and more of these new rule types for data quality monitoring.

Following we will show some existing solutions for data quality monitoring, starting with the predecessor of the project IQM4HD, the Data Checking Engine (DCE) [18]. It is an SQL-based system for complex data quality rules. The DCE provides an interface for data quality managers to compose data quality rules in form of SQL statements. Even though templates were introduced to reduce the complexity of rule composition, especially regarding the creation of multiple rules of the same type for a lot of different data, the complexity of SQL and the underlying database schema still has to be known by the domain expert. Also, once written, rules were hard to read and to fully comprehend. Therefore, this project, as a successor, and especially the DSL copes with the mentioned problems the DCE has, in form of the separation of concerns via the split into Sources, Checks and Actions.

Endler et al. developed an architecture for continuous data quality monitoring in medical centers [12] which has many similarities with our proposal. They also handle data quality monitoring with a rule-based approach. In contrast to our solution, they did not introduce a DSL for describing rules. The most basic shown rules are simple boolean operations directly defined for the underlying database. That also shows the second part where our approaches differ, the rule system from Endler et al. is designed for a special medical unified database system without the possibility to easily extend the database environment or adapt other schemas.

Another approach for a rule based data validation system called GuardianIQ is introduced by David Loshin [23]. His concept is a quality monitoring based on business rules and executed in SQL. Focus of this solution also lies on the distinction between data cleansing and data quality in conjunction with metrics to quantify data quality. However, we were not able to perform a closer comparison, since only rough information was given in a short paper from 2002 without newer publications.

Lastly, Batini et al. suggested a methodology for data quality assessment and improvement called Heterogenous Data Quality Methodology (HDQM) [5]. As the name suggests, their idea is a methodology applied to the processes of businesses. It provides guidelines - in form of a so called meta-model - on how to design the data quality process integrated in the target system. Whereas their system integrates in the business process requiring a complete remodeling of the users system, our approach can be seen as a complementary addition to apply to existing systems.

All in all, partial solutions similar to our approach exist, especially when looking at the DCE or the rule based approach from Endler et al. What is missing is a unified framework to cover all needs for data quality monitoring including most techniques for data quality checking and the ability to easily extend them, together with - especially - the aspect of monitoring heterogeneous data sources, which has neither been covered by the DCE, nor Endler's solution. That is why the extension of compatibility for heterogeneous data and the split between technical and subject-specific knowledge sets our approach apart from existing solutions.

7. Conclusion

7.1. Open Issues

- The DSL has lots of reserved words, that must not be used as identifiers. This currently includes all names of build-in methods. This problem is not so bad as we decided to make the DSL case-sensitive, and all reserved words must be in upper case. However, a better handling of e.g. method names would be better.
- The implementation of list expressions currently only supports equijoins.
- In case of errors, only the predefined roles are reported. Often, additional roles are useful to identify the data error. Thus a new predefined role to report additional information evaluated by the RETURN statement would be helpful. Another solution would be to always report any role present in the return statement.
- The implementation of aggregation functions (e.g. SUM) does not allow expressions as parameters, only single identifiers. This leads to over-complicated formulations of checks. An example is the DistributionCheck.
- Currently, the role assignment from sources proceeds from the existing attributes in the source. Thus, for each attribute that is present, a role definition is searched. If one is found, then a corresponding role is created in the record. This means that a missing attribute does not lead to a role that contains a NULL value but rather in a missing role. To compensate, the interpreter assumes NULL when a non-present role is accessed. However, this makes finding errors in the DSL code much more complicated, as a misspelled role name is substituted silently with NULL.
- The error handling is very bare; it is often complicated to find errors in self-written DSL code due to incomprehensible error messages.
- The PERCENTAGE keyword is not implemented
- The IF construct in EXECUTE-statements in ACTIONS is not implemented.
- Within a check, no Source can be addressed. This could be helpful, as an example, we have the EmailCheck that references the email pattern from within the check. However, this would need some additional syntax that disambiguates role / variable names and source names.

- We have a separate prototype that optimizes the access order of data sources when running multiple profiling modules in parallel. This must be integrated with the main prototype.
- The handling of multiple consecutive errors in time series models does not work currently.

A. Rulecatalog

A.1. Column property checks

These checks target individual values of single attributes or single records. Each Check will be described and an example Action will be given. The Sources used in those examples are the following:

```
SOURCE Customer TYPE LIST QUERY ROLES (id: IDENTIFIER):
DATABASE cust NATIVE
  SELECT id, name, firstname, dob, email,
                title, bonus
  FROM customer
END
```

```
SOURCE EmailPattern TYPE CONST ROLES (email):
  "[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}"
END
```

A.1.1. NotNullCheck

The NotNullCheck is one of the most simple and basic checks this software provides. Its return value is dependent on whether the given value is null or not. This should only be used if you are certain the data set you want to check does not have many null values. Too much error output would occur otherwise.

Note hat due to the execution logic, if a compound role is given as value, all parts must be NULL to count as a violation, because

```
[13,NULL] IS NULL = [FALSE,TRUE] = FALSE
```

and thus no TRUE is returned. If it is desired to find such situations, a NULL check must be applied individually to each attribute.

- *value* Value to be checked (a record, might consist of multiple attribute)

```
CHECK NotNullCheck ON value:
  RETURN value IS NULL;
END
```

The following Action uses the NotNullCheck to check the customer's email addresses for NULL values. The result set will mark all tuples of the customer table where the email field is NULL as an error.

```
ACTION NotNullCheckCustomerEmail:
  EXECUTE NotNullCheck ON EACH Customer(email)
  RESULT IN ERROR
END
```

A NotNullCheck may also be used on a key inside a document originating from a document database as shown in the following listing.

```
SOURCE CustomerAddressWithEmpty TYPE LIST QUERY ROLES
  (_id: IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.aggregate([
    { $unwind: { path : "$address",
                 preserveNullAndEmptyArrays: true } } ])
END
```

```
ACTION NotNullCheckCustomerAddress:
  EXECUTE NotNullCheck
    ON EACH CustomerAddressWithEmpty(address)
    RESULT IN ERROR
END
```

A.1.2. RangeCheck

The RangeCheck is another very basic Check. It checks whether the given value is in the defined range. When the given record contains more than one attribute, each attribute has to be in the range in order to not violate the Check.

- `ON` *value* - Value to be checked
- `WITH` *lower, upper* - range thresholds the value should be checked against

```
CHECK RangeCheck ON value WITH lower, upper:
  RETURN NOT (value >= lower AND value <= upper);
END
```

This Action checks whether the customers age's are within the range of a person of age (18) as the lower bound and 100 as the upper bound.

```
ACTION RangeCheckCustomerAge:
  EXECUTE RangeCheck ON EACH CustomerWithAge(age) WITH 18,
     ↪ 100
```

```
  RESULT IN ERROR
END
```

A.1.3. BoundaryChecks

There are two boundary Checks; Lesser and Greater. They can be used if the range of values is bound only in one way. For example a LesserCheck could be applied to the price column of a product table. A product should not cost less than 0.01.

- `ON` *value* - Value to be checked
- `WITH` *threshold* - threshold the value should be checked against

```
CHECK NotLesserCheck ON value WITH threshold:
  RETURN value < threshold;
END
```

```
CHECK NotGreaterCheck ON value WITH threshold:
  RETURN value > threshold;
END
```

```
ACTION NotLesserCheckCustomerAge:
  EXECUTE NotLesserCheck ON EACH CustomerWithAge(age) WITH
     ↪ 18
  RESULT IN ERROR
END
```

A.1.4. PatternCheck

The PatternCheck matches the given value against a regular expression.

- *value* - Value to be checked
- *pat* - A regular expression describing valid *values*

```
CHECK PatternCheck ON value WITH pat:
  RETURN NOT MATCHES(value, pat);
END
```

A special case of the PatternCheck is the EmailCheck. It was introduced due to the fact, that email addresses are a very common and always follow the same pattern.

- *email* - Email address which should be checked for the correct syntax

```
CHECK EmailCheck ON email:
  EXECUTE PatternCheck ON email WITH EmailPattern;
END
```

Another special case is when the source data for the pattern check is found inside a nested document; in this case the phone numbers should match a phone number pattern. This is primarily handled by a fitting Source declaration:

```
SOURCE CustomerPhoneProj TYPE LIST QUERY ROLES
  (_id: IDENTIFIER, phonePos: IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.aggregate( [
    { $unwind : { "path" : "$phone",
                  includeArrayIndex : "phonePos" } },
    { $project : { "name" : 1, "firstname" : 1,
                   "phone" : 1, "phonePos" : 1 } }
  ] )
END
```

The Check and Action are similar to above (the definition of `PhonePattern` is straight forward and omitted here):

```
ACTION PhonePatternCheck:
  EXECUTE PatternCheck
  ON EACH CustomerPhoneProj(phone: value)
  WITH PhonePattern
  RESULT IN ERROR
END
```

If additionally only certain elements have to match a given pattern, this can be expressed by adjusting the Check accordingly; in this example every second phone number per customer has to be a mobile phone number which can be identified by a specific pattern `MobilePhonePattern`:

```
CHECK PositionPatternCheck ON LIST values(val, pos) WITH
   ↪ pat:
  LIST nos := SELECT val FROM values WHERE pos == 2;
  FOR n IN nos:
    RETURN NOT MATCHES(n, pat);
  END
END
```

```
ACTION MobilePhonePatternCheck:
  EXECUTE PositionPatternCheck
  ON CustomerPhoneProj(phone: val, phonePos: pos)
  WITH MobilePhonePattern
```

```
  RESULT IN ERROR
END
```

Note, that the Check specification is completely independent of the application area and the same check may also be used to check a pattern in the second element of e.g. a list of first names by calling it from a different Action. In addition, the implementation should be made even more generic in practice by supplying the position to be used for the Check as an additional parameter; this is omitted here for simplicity of presentation.

A.1.5. ContainsCheck

- *value* - column value to be checked (or columns)
- *values* - the set the *value* should be contained in

```
CHECK ContainsCheck ON val WITH LIST values:
  RETURN val IN values;
END
```

A.1.6. NestedDocumentValueCheck

The following example can only be applied to Sources which provide hierarchically nested values as data such as document databases. The example shows how it can be checked that every `address` named value in a list element contains a sub-element `street` which in turn contains a sub-element with key `name` which is not empty.

```
SOURCE CustAddWithPos TYPE LIST QUERY
                      ROLES (_id: IDENTIFIER,
                             adrPos: IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.aggregate([
    { $unwind: { path : "$address",
                 includeArrayIndex : "adrPos" } } ] )
END
```

```
ACTION NotNullCheckCustomerCountry:
  EXECUTE NotNullCheck
  ON EACH CustomerAddress(address.country: value)
  RESULT IN ERROR
END
```

A.2. Structure analysis checks

A.2.1. QuantityCheck

```
CHECK QuantityCheck ON LIST list1 WITH LIST list2:
  LIST cnt1 := SELECT COUNT(*) FROM list1;
  LIST cnt2 := SELECT COUNT(*) FROM list2;
  RETURN cnt1 != cnt2;
END
```

```
ACTION DWHQuantityCheckCustomer:
  EXECUTE QuantityCheck ON DWH_Customer WITH Customer
  RESULT IN ERROR
END
```

A.2.2. SumCheck

```
CHECK SumCheck ON LIST list1(val1) WITH LIST list2(val2):
  LIST sum1 := SELECT SUM(val1) FROM list1;
  LIST sum2 := SELECT SUM(val2) FROM list2;
  RETURN sum1 != sum2;
END
```

```
ACTION DWHSumCheckCustomer:
  EXECUTE SumCheck ON DWH_Customer(bonus) WITH Customer(
     ↪ bonus)
  RESULT IN ERROR
END
```

A.2.3. UniqueCheck

```
CHECK UniqueCheck ON LIST src(val):
  RETURN LIST
    SELECT val ROLE value, COUNT(*) ROLE cnt
    FROM src
    GROUP BY val
    HAVING cnt > 1;
END
```

A.2.4. Referential Integrity

```
CHECK ReferentialIntegrityCheck ON LIST src(fk)
                                WITH LIST target(pk):
  RETURN LIST
    SELECT fk ROLE value
    FROM src LEFT JOIN target ON fk = pk
    WHERE pk IS NULL;
END
```

The referential integrity check also works with nested values, e. g. originating from document databases as shown in the following example: There the nested document with key `family` which is a list of family members of a customer is equipped with the check that `name` and `firstname` have to refer to an existing customer document:

```
SOURCE CustomerFamilyWithPos TYPE LIST QUERY ROLES
  (_id: IDENTIFIER, familyPos: IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.aggregate([
    { $unwind: { path : "$family",
                 includeArrayIndex : "familyPos" } }
  ] )
END
```

```
ACTION RICheckCustomerFamily:
  EXECUTE ReferentialIntegrityCheck
  ON CustomerFamilyWithPos(family.name: fk,
                           family.firstname: fk)
  WITH CustomerFamilyWithPos(name: pk, firstname: pk)
  RESULT IN ERROR
END
```

A.2.5. Functional Dependency

```
CHECK FunctionalDependencyCheck ON LIST src(left, right):
  RETURN LIST
    SELECT left ROLE value,
           MIN(right) ROLE minval, MAX(right) ROLE maxval
    FROM src
    GROUP BY left
    HAVING minval != maxval;
END
```

The functional dependency check can also be applied to nested values. In the example below a functional dependency from ZIP code to city is assumed, but the information is only available nested inside the `address` sub-document of the customers:

```
SOURCE CustAddWithPos TYPE LIST QUERY
                      ROLES (_id: IDENTIFIER,
                             adrPos: IDENTIFIER):
DATABASE custmongo NATIVE
  db.customer.aggregate([
    { $unwind: { path : "$address",
                 includeArrayIndex : "adrPos" } } ] )
END
```

```
ACTION FdCheckCustomer:
  EXECUTE FunctionalDependencyCheck
  ON CustAddWithPos(address.zip: left, address.city: right)
  RESULT IN ERROR
END
```

A.3. Value checks

A.3.1. DistributionCheck

- *set* -
- *refDistribution* -a reference distribution the original set will be calculated by and then ch

```
CHECK DistributionCheck ON LIST data(val)
      WITH refDistribution(from, to, bin, perc):
  LIST cnt := SELECT COUNT(*) FROM data;
  LIST bindata := SELECT VAL2BIN(val, refDistribution) ROLE
     ↪  bval
             FROM data;
  LIST bincnt := SELECT bval ROLE bval, COUNT(*) ROLE acnt
            FROM bindata
            GROUP BY bval;
  LIST hist :=
    SELECT bin, perc, NVL(acnt, 0) ROLE acnt
    FROM bincnt RIGHT JOIN refDistribution ON bval = bin;
  LIST score1 :=
    SELECT (acnt - cnt * perc) * (acnt - cnt * perc) /
           (cnt * perc) ROLE cell
    FROM hist;
```

```
  LIST score :=
    SELECT SUM(cell) ROLE score FROM score1;
  RETURN LIST score;
END
```

A.3.2. ARCheck

An AR check is a time series check that models the time series using an AR model. The basic formula is as follows:

$$\hat{y}_t = y_{t-1}a_1 + \cdots + y_{t-n}a_n \tag{A.1}$$

```
CHECK ARCheck ON LIST ts(time, value) WITH LIST arparam, sd
   ↪ :
  LIST ts := SELECT * FROM ts ORDER BY time;
  FOR t := LENGTH(arparam)+1 TO LENGTH(ts):
    yt := 0;
    FOR i := 1 TO LENGTH(arparam):
      yt := yt + ts[t-i].value * arparam[i];
    END
    score := ABS(ts[t].value - yt) / sd;
    RETURN score ROLE score, ts[t].time ROLE identifier;
  END
END
```

A sample source for the parameter will look like this:

```
SOURCE ARParam TYPE LIST CONST ROLES (ar):
  .50,
  .25,
  .1275
END
```

A time series data source might look like this:

```
SOURCE TimeSeriesSales TYPE LIST QUERY
                       ROLES(time_id: IDENTIFIER):
DATABASE ts NATIVE
  SELECT '' || week || '-' || day as time_id,
         sales_volume
  FROM time_series_sales
END
```

With these source definitions, we can call the check as follows:

```
ACTION ARCheckTsSales:
  EXECUTE ARCheck ON TimeSeriesSales(time_id, sales_volume)
  WITH ARParam, 10
  RESULT IN WARNING ABOVE 2 AND ERROR ABOVE 3
END
```

A.3.3. CubeCheck

```
CHECK CubeCheck ON LIST cube(time, dim, metric)
                   WITH LIST cubemodel:
  RETURN LIST CUBESCORE(cube, cubemodel);
END
```

Bibliography

[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: A survey. The VLDB Journal, 24(4):557–581, Aug. 2015.

[2] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In Proceedings of the 22th International Conference on Very Large Data Bases, VLDB '96, pages 506–521, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[3] C. C. Aggarwal. Outlier Analysis. Springer New York, 1 edition, 2013.

[4] P. Atzeni, F. Bugiotti, and L. Rossi. Sos (save our systems): A uniform programming interface for non-relational systems. In Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, pages 582–585, New York, NY, USA, 2012. ACM.

[5] C. Batini, D. Barone, F. Cabitza, and S. Grega. A Data Quality Methodology for Heterogeneous Data. International Journal of Database Management Systems, 3(1):60–79, Feb. 2011.

[6] C. Binnig, R. Rehrmann, F. Faerber, and R. Riewe. Funsql: It is time to make sql functional. In Proceedings of the 2012 Joint EDBT/ICDT Workshops, EDBT-ICDT '12, pages 41–46, New York, NY, USA, 2012. ACM.

[7] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In 2007 IEEE 23rd International Conference on Data Engineering, pages 746–755. IEEE, 2007.

[8] L. Bravo, W. Fan, F. Geerts, and S. Ma. Increasing the expressivity of conditional functional dependencies without extra complexity. In 2008 IEEE 24th International Conference on Data Engineering, pages 516–525. IEEE, 2008.

[9] P. Buneman, M. Fernandez, and D. Suciu. Unql: a query language and algebra for semistructured data based on structural recursion. The VLDB Journal, 9(1):76–110, Mar 2000.

[10] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13, pages 403–416, New York, NY, USA, 2013. ACM.

[11] F. Chiang and R. J. Miller. Discovering data quality rules. Proceedings of the VLDB Endowment, 1(1):1166–1177, 2008.

[12] G. Endler, P. K. Schwab, A. M. Wahl, J. Tenschert, and R. Lenz. An architecture for continuous data quality monitoring in medical centers. MEDINFO, 2015.

[13] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering Conditional Functional Dependencies. IEEE Transactions on Knowledge and Data Engineering, 23(5):683–698, May 2011.

[14] S. Feuerstein and B. Pribyl. Oracle pl/sql Programming. " O'Reilly Media, Inc.", 2005.

[15] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The tsimmis approach to mediation: Data models and languages. Journal of Intelligent Information Systems, 8(2):117–132, Mar 1997.

[16] M. Gupta, J. Gao, C. Aggarwal, and J. Han. Outlier Detection for Temporal Data: A Survey. Knowledge and Data Engineering, IEEE Transactions on, 26(9):2250–2267, Sept. 2014.

[17] F. Heine. Outlier Detection in Data Streams Using OLAP Cubes. In New Trends in Databases and Information Systems, Communications in Computer and Information Science, pages 29–36. Springer, Cham, Sept. 2017.

[18] F. Heine, C. Kleiner, A. Koschel, and J. Westermayer. The Data Checking Engine: Complex Rules for Data Quality Monitoring. 2014.

[19] F. Heine and M. Rohde. PopUp-Cubing: An Algorithm to Efficiently Use Iceberg Cubes in Data Streams. In Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, BDCAT '17, pages 11–20, New York, NY, USA, 2017. ACM.

[20] R. J. Hyndman and G. Athanasopoulos. Forecasting: principles and practice. OTexts, 2018.

[21] R. Lawrence. Integration and virtualization of relational sql and nosql systems including mysql and mongodb. In 2014 International Conference on Computational Science and Computational Intelligence, volume 1, pages 285–290, March 2014.

[22] X. Li and J. Han. Mining approximate top-k subspace anomalies in multi-dimensional time-series data. In Proceedings of the 33rd international conference on Very large data bases, pages 447–458. VLDB Endowment, 2007.

[23] D. Loshin. Rule-based data quality. In Proceedings of the eleventh international conference on Information and knowledge management, pages 614–616. ACM, 2002.

[24] J. E. Olson. Data Quality: The Accuracy Dimension. Morgan Kaufmann, Jan. 2003.

[25] L. L. Pipino, Y. W. Lee, and R. Y. Wang. Data Quality Assessment. Commun. ACM, 45(4):211–218, Apr. 2002.

[26] J. Rith, P. S. Lehmayr, and K. Meyer-Wegener. Speaking in tongues: Sql access to nosql systems. In Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, pages 855–857, New York, NY, USA, 2014. ACM.

[27] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger. Automating large-scale data quality verification. Proceedings of the VLDB Endowment, 11(12):1781–1794, Aug. 2018.

[28] S. Sobieski and B. Zieliński. Using maude rewriting system to modularize and extend sql. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, pages 853–858, New York, NY, USA, 2013. ACM.

[29] Y. Wand and R. Y. Wang. Anchoring Data Quality Dimensions in Ontological Foundations. Commun. ACM, 39(11):86–95, Nov. 1996.

[30] R. Y. Wang, V. C. Storey, and C. P. Firth. A framework for analysis of data quality research. IEEE transactions on knowledge and data engineering, 7(4):623–640, 1995.

[31] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In Proceedings of the 29th international conference on Very large data bases-Volume 29, pages 476–487. VLDB Endowment, 2003.

[32] C. Zhang and J. Xu. A unified sql middleware for nosql databases. In Proceedings of the 2018 International Conference on Big Data and Computing, ICBDC '18, pages 14–19, New York, NY, USA, 2018. ACM.