The Data Checking Engine: Monitoring Data Quality

Felix Heine, Carsten Kleiner, Arne Koschel
University of Applied Sciences & Arts Hannover
Faculty IV, Deptartment of Computer Science, Hannover, Germany
Email: firstname.lastname@hs-hannover.de

Jörg Westermayer
SHS Viveon
Germany
Email: joerg.westermayer@shs-viveon.de

***Abstract*—In the context of data warehousing and business intelligence, data quality is of utmost importance. However, many mid-size data warehouse (DWH) projects do not implement a proper data quality process due to huge up-front investments. However, assessing and monitoring data quality is necessary to establish confidence in the DWH data. In this paper, we describe a data quality monitoring system developed collaboratively at HS Hannover and SHS Viveon: The "Data Checking Engine" (DCE). The goal of the system is to provide DWH projects with an easy and quickly deployable solution to assess data quality while still providing highest flexibility in the definition of the assessment rules. It allows to express complex quality rules and implements a template mechanism to facilitate the deployment of large numbers of similar rules.**

***Keywords*—Data Quality, Quality Rules, Data Analysis, Data Quality Monitoring, Data Warehouses**

I. Introduction

Data quality (DQ) is of utmost importance for a successful data warehouse project. In this context, continuous monitoring is an integral part of any DQ initiative. In this paper, we describe a data quality monitoring system called *Data Checking Engine* (DCE) developed collaboratively at the University of Applied Sciences & Arts Hannover and SHS Viveon. The main goal is to provide a flexible, yet simple tool to monitor data quality in DWH projects, which can also be used during the DWH development to test its Extract Transform Load (ETL) process.

To constantly monitor the quality of data of a database, it is necessary to define quality rules using a flexible rule definition language. Quality rules are either derived from business rules or found via profiling or data mining. They are executed either in regular intervals or based on specific events like the completion of an ETL job. The results of the rule runs are recorded in a result repository, which also keeps historical data so that users can evaluate the quality of data over time. As rules will evolve over time, it is necessary to keep a history of rule definitions so that historic results can be related to the correct version of the rule's definition.

We believe the ability to express complex rules is crucial. A set of hard-coded rule types found in some data quality tools is typically only suitable to detect rather simple quality problems on the attribute or single tuple level. However, there are more complex data quality problems, which cannot be detected using such rules. As an example, consider an error located in the logic of an ETL process. Due to this error, the process fails to reference the correct product group for some of the records of a sales fact cube. The bug is subtle and does not show up very often. At the attribute level all sales records are correct. However, the trend of the time series showing the sales sum with respect to individual product groups will indicate a quality problem.

It requires skilled users to write such rules but larger sets of rules will look similar in structure. They differ only in the tables and attributes they are applied to. For this, a template mechanism is useful to help users define such rules. The idea is that only the template writer has to cope with the full complexity; template users can then apply these templates to their tables and attributes.

To avoid discontinuity of the reporting environment for DWH users, re-using existing Business Intelligence (BI) tools is superior over building a specialized quality reporting GUI. Still it is sufficient to export rule results to a quality data mart, which can then be accessed by any standard BI tool. However, the plain rule results have to be aggregated to more comprehensive quality metrics in a flexible and user defined way.

Furthermore, the rules themselves have to be tested in the development environment before deployment. Thus, an automated transfer and synchronization with the production system is necessary.

In a nutshell, we target the following requirements:

- Express complex rules
- Reduce complexity of rules (utilizing a template mechanism)
- Execute the rules regularly or upon specific events
- Keep a history of rule definitions and execution results
- Store this history in a quality data mart persistently
- Aggregate the rule results to quality metrics
- Provide export/import mechanism for rule meta data

The remainder of this paper is organized as follows: In the following section, we give an overview of related work. Section III focuses on the definition of quality rules and explains our template mechanism. Section IV describes the DCE architecture. In the subsequent section, we briefly elaborate on quality metrics. In the final section, we summarize our achievements and give an outlook to our future plans.

II. Related Work

Over the last decade, much research in the data quality domain has been conducted, see for example [1]. Research areas related to data quality are outlier detection, data deduplication, data quality monitoring, data cleansing, and data mining to detect quality rules. We are specifically interested in monitoring and reporting data quality. In general, we follow the approach of Kimball [2] who outlines an approach to DQ assessment in DWH systems.

For our work, we are especially interested in formalisms to describe quality rules. Most existing approaches target only specific types of rules. Edit rules describe invalid tuples [3]. In [4], editing rules are used to match records with master data. In [5], *conditional functional dependencies* are used to express more complex rules spanning multiple tuples of a relation. The same book also describes *conditional inclusion dependencies* that are generalizations of referential integrity checks. These approaches can be reformulated to SQL, thus DCE is able to execute such rules. Another type of rules are differential dependencies, see [6].

In the domain of data deduplication (also called record linkage), rules are important to describe matching criteria. As an example, the IntelliClean [7] system uses rules like `<if> condition <then> action` *`with probability p`* to match duplicates. Currently, we do not target this issue, although we plan to integrate these features in our system in the future.

Another approach is to extend SQL to incorporate data quality features. An example is the FraQL [8] language that specifies pivoting features and allows to integrate user defined grouping and aggregate functions that allow to analyze data more comfortably. The drawback is that a special execution engine is required. Thus, the features of existing relational optimizers are not available or have to be reproduced.

Furthermore, many prototypic research systems and commercial tools are present. For an overview, see [9]. Most existing tools focus on dimension data only and thus stress single record problems and deduplication.

However, to the best of our knowledge, no tool provides a similar mechanism that allows to build complex rule templates, which can, for example, be used to test indicator values against time series models.

III. Rule Definition Language

A central issue is the language to define the quality rules. On one hand, it has to be expressive to allow for complex rules like time series tests. On the other hand, fast definitions of simple rules like NULL value checks has to be possible. Also, the rule execution is typically critical with respect to execution time and resource consumption. As large datasets have to be checked, an efficient rule execution engine is demanded.

Thus, we decided to rely on the native SQL executor of the DBMS. This means, the core of each rule is an SQL statement, which collects the required information from the underlying tables. This statement is written by the DCE user, allowing even vendor-specific optimizations like optimizer hints.

DCE defines a *standard attribute set* for the result tuples. The rule statements have to adhere to this standard. Each statement computes a *result value*, which is the basis for the rule check. For a NULL rule, the result value might be the percentage of NULL values of the checked values. There might either be a single result value or multiple values, broken down by dimensional hierarchies. The latter case might for example yield a percentage of NULL values for each product group in each region. For each rule, *multiple bounds* can be defined, specifying valid ranges for the observed values. The bounds can be activated or deactivated with respect to all values contained in the result tuple. In this way, the bound for NULL values can be normally defined to be 5 percent, however for specific product groups it might be higher. A specific application for this feature is to change bounds for business metrics, e.g., according to the week day. Typically, the revenue sum for traditional stores might be zero on Sundays.

A *severity* can be assigned to each rule bound, and multiple bounds with different severity can be defined for a rule. The severity information of failed rules is returned to the scheduler. Based on this information, the scheduler might, e.g., decide to interrupt an ETL process or to alert the DWH team.

Each rule's SQL statement can have *multiple parameters*, which are set at execution time. These parameters can for example be used to determine the range of data to be checked. In this way, a quality rule running after an ETL job might be limited to check only the new records in a fact table.

In the following, we describe a *sample rule*. The basic idea of the example is to test indicator values like revenue stored in a fact table on a regular basis against a moving average of its past values. For simplicity, we assume there is no seasonal component, although this is not a limit of the system. The following formula describes our model:

$$Y_t = \frac{1}{k}\sum_{d=1}^{k} Y_{t-d} + \epsilon_t \qquad (1)$$

Here, ϵ_t is a random component, which is Gaussian with $\mu = 0$ and some σ. The trend is based on the moving average of the k previous values of the indicator.

During DWH operation, we assume that past values for the indicator are already checked and corrected. Each day after the ETL process has finished, we want to test the new value. Thus, we have to calculate

$$k_t = Y_t - \left(\frac{1}{k}\sum_{d=1}^{k} Y_{t-d}\right) \qquad (2)$$

and then check whether k_t is within a certain interval. As ϵ_t is Gaussian, we know that 95% of the values will be within the interval $[-2\sigma, 2\sigma]$. We could use these bounds to generate a warning and the $[-3\sigma, 3\sigma]$ interval to generate an error.

```
SELECT today.day day,
       avg(today.revenue) -
       avg(past.revenue) result
FROM sales_fact today, sales_fact past
WHERE today.day = $testday$
  AND (today.day - past.day)
      BETWEEN 1 AND k
GROUP BY today.day
```

Fig. 1. Sample rule code (simplified)

A simplified SQL for these checks is shown in Fig. 1. The statement returns a result value k_t, which is then checked by the DCE against bounds like the $[-2\sigma, 2\sigma]$ interval. This statement has a parameter `$testday$`, which is replaced at runtime with the current day.

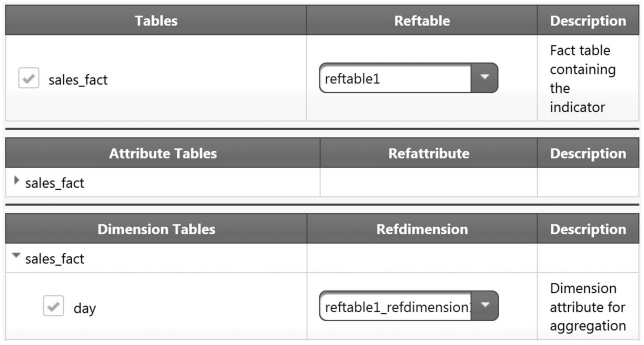
In typical environments, there is often a need to define a number of equivalent rules over a large number of tables and attributes. To accommodate for this requirement, we implemented a *template concept*.

A template looks quite similar to a normal rule. It contains a SQL statement producing the same set of standard columns, and it might also contain bound definitions. However, instead of the target table and attribute names, the template's SQL statement contains special markers. For attributes, these markers declare the purpose of the attribute within the rule. Once the user has defined a template, she can instantiate it for multiple sets of tables and attributes. During this process, she either defines new bounds or uses the predefined bounds from the template for the generated rules. The engine forwards rule parameters defined within the template to the generated rules.

```
SELECT today.§refdim1§ day,
       avg(today.§refattr1§) -
       avg(past.§refattr1§) result
FROM §reftable1§ today, §reftable1§ past
WHERE today.§refdim1§ = $testday$
  AND (today.§refdim1§ - past.§refdim1§)
      BETWEEN 1 AND k
GROUP BY today.§refdim1§
```

Fig. 2. Sample template code (simplified)

The *sample* statement is a good candidate for a template. In the template, there is another type of parameters called template parameters that are replaced at template instantiation (i.e., rule creation time). These are used to define placeholders for the table and attribute names, like §reftable1§ (cf. Fig. 2).



Tables	Reftable	Description
✓ sales_fact	reftable1	Fact table containing the indicator

Attribute Tables	Refattribute	Description
sales_fact		

Dimension Tables	Refdimension	Description
sales_fact		
✓ day	reftable1_refdimension	Dimension attribute for aggregation

Fig. 3. Instantiating a template

A *GUI* assists unexperienced users with defining the template parameters, as shown in Fig. 3. For this dialog, the GUI reads the database catalog and lets the user map the template parameters to catalog objects. E.g., §reftable1§ is replaced with `sales_fact`.

IV. Architecture

Figure 4 shows an overview of the DCE overall architecture. The DCE itself is organized as a classical three-tier application. It interacts with the enterprise data warehouse system in order to compute quality indicators. Also, results of the data quality checks may be propagated into another external database system, the data quality data mart. This database in itself is also organized as a data mart and provides long term storage of computed data quality indicators in order to be used for long term analysis of enterprise wide data quality. In a sense it is a meta-data warehouse for data quality. There is also an external scheduling component (typically standard system scheduling capabilities), which triggers computation of data quality indicators at previously defined points in time.

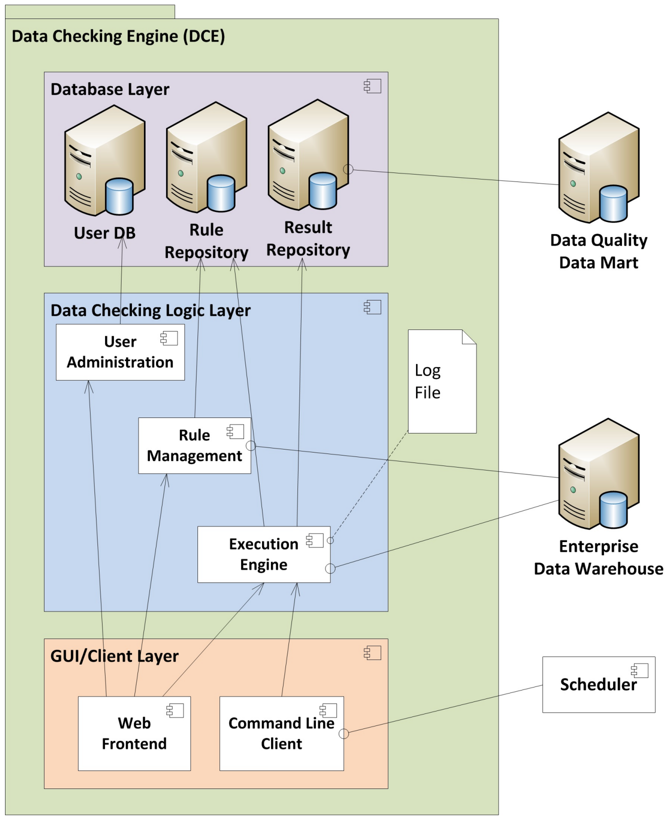


Fig. 4. Data checking engine architecture overview

Within the DCE itself the main entry point for data quality managers is the GUI of the DCE web application (shown at the bottom of Fig. 4). The GUI is used to manage users of the DCE application, to manage data quality rules, and to manage data rule executions. As typically the execution of data quality checks is not triggered manually, there is also a command-line client library for the rule execution engine that is triggered by an external scheduler. The schedule to be used is managed in the web application as well.

The main data checking business logic can be found in the middle tier. This logic is used by the web application as described above. Note that there is a strict separation between user management, rule management and rule execution management in the middle tier as well. Whereas the user administration component provides standard functionality, note that the rule management component contains advanced features. For instance the template mechanism described in the previous section is implemented here.

The execution engine is also managed by the web application: on one hand rules can be manually executed from the web application, on the other hand scheduled execution can be defined here.

During rule execution, the engine replaces the parameters in the rule's SQL statement with their current values and then runs the statement using the target database. Thus, moving large amounts of data into the DCE engine is avoided. The result of the SQL statement is then further processed. This includes checking the currently applicable bounds and testing their severity.

In the execution engine it is also defined, which rules are executed on what data warehouse database under whose privileges. Note that multiple different data warehouses (or database systems) may be used as source, because the connection information is also managed by the web application.

Finally the database layer consists of three separate areas:

- Rule repository, which holds the data quality rules as well as base templates
- Result repository holding results of rule execution
- User database which is used for access management to only the DCE itself

Once results of the executed data quality rules have been stored in the result repository they may be propagated to the data quality data mart that aggregates the results into quality indicators.

This data mart is not part of the DCE but located within the standard DWH infrastructure of the company. Thus, standard interfaces such as reporting and BI tools can be used to further present and analyze the data quality status. This way the additional effort for data quality monitoring can be kept minimal as access to data quality indicators follows well established processes and uses well-known tools, which are used for regular monitoring of enterprise performance indicators as well. In addition, the concept of viewing data quality indicators similarly as regular performance indicators is very fitting, as these have to be tracked accordingly in order to ensure reliability of data in the data warehouse. Ultimately, this is necessary to make the right entrepreneurial decisions based on reliable information.

V. Data Quality Metrics

The result repository contains large amounts of specific results that individually describe only a very small fraction of the overall data quality of the DWH. In order to get a quick overview of the quality level, a small set of metrics that aggregate the rule results is required.

In the literature, there are various approaches to define data quality indicators, for example [10]. Thus we decided to provide a flexible approach that enables the user to define her own indicator hierarchies. The engine stores indicator definition meta data and calculates the resulting indicator values.

An important issue here is to take incremental checks into account. As an example, consider a rule that checks the number of dimension foreign keys in a fact table that reference a dummy instead of a real dimension entry. As the fact table is large, the daily rule just checks the new fact records loaded in the previous ETL run. Thus the indicator has to aggregate over the current and past runs to provide an overall view of the completeness of the dimension values.

VI. Conclusion and Future Work

We have already implemented a prototypical version of the described architecture. Furthermore, we have validated the approach by interviewing teams of different DWH projects and building project specific prototypical setups. Our engine has been able to support their quality monitoring requirements. Especially the flexibility in rule definition was appreciated. We have not only detected quality problems on tuple level but also more complex issues, e.g., checking the trend of indicators stored in a fact table. As expected, our template mechanism has proven to be an important way to simplify rule definition.

The engine keeps a comprehensive history of rule results *and* rule meta data, which allows to monitor data quality over time and to check whether quality improvement projects were successful. This quality data is exposed to external BI tools for reporting and further analysis.

An important consequence of the flexibility of our approach is that the DCE can also be used during DWH/ETL development to test the result processes. The testing rules developed during this project phase may also be used during normal operation, reducing the overall cost.

Our approach is currently working on any relational database system. In the future, we plan to also integrate Big Data systems like Hadoop, as more and more relevant data is stored in such systems. Thus data quality should be monitored there as well. As there is currently no universal query language standard like SQL in the relational sector, we will have to devise a flexible way to cope with various rule definition languages.

References

[1] C. Batini and M. Scannapieco, *Data Quality: Concepts, Methodologies and Techniques*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[2] R. Kimball and J. Caserta, *The data warehouse ETL toolkit*. Wiley, 2004.

[3] I. P. Fellegi and D. Holt, "A systematic approach to automatic edit and imputation," *Journal of the American Statistical Association*, vol. 71, no. 353, pp. 17–35, 1976.

[4] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu, "Towards certain fixes with editing rules and master data," *The VLDB Journal*, vol. 21, no. 2, pp. 213–238, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1007/s00778-011-0253-7

[5] W. Fan and F. Geerts, *Foundations of Data Quality Management*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[6] S. Song and L. Chen, "Differential dependencies: Reasoning and discovery," *ACM Trans. Database Syst.*, vol. 36, no. 3, pp. 16:1–16:41, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000824.2000826

[7] M. L. Lee, T. W. Ling, and W. L. Low, "Intelliclean: A knowledge-based intelligent data cleaner," in *ACM SIGKDD, Boston, 2000*, 2000.

[8] K. Sattler, S. Conrad, and G. Saake, "Adding conflict resolution features to a query language for database federations," in *Proc. 3nd Int. Workshop on Engineering Federated Information Systems, EFIS'00, Dublin, Ireland, June*, 2000, pp. 41–52.

[9] J. Barateiro and H. Galhardas, "A survey of data quality tools," *Datenbank-Spektrum*, vol. 14, 2005.

[10] B. Heinrich, M. Kaiser, and M. Klier, "Metrics for measuring data quality - foundations for an economic oriented management of data quality," in *Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT). INSTICC/Polytechnic Institute of Setúbal*, J. Filipe, B. Shishkov, and M. Helfert, Eds., 2007.