CM-Explorer: Dissecting Data Ingestion Problems

Niels Bylois
UHasselt
Data Science Institute, ACSL
Belgium
niels.bylois@uhasselt.be

Frank Neven
UHasselt
Data Science Institute, ACSL
Belgium
frank.neven@uhasselt.be

Stijn Vansummeren
UHasselt
Data Science Institute, ACSL
Belgium
stijn.vansummeren@uhasselt.be

ABSTRACT

Data ingestion validation, the task of certifying the quality of continuously collected data, is crucial to ensure trustworthiness of analytics insights. A widely used approach for validating data quality is to specify, either manually or automatically, so-called data unit tests that check whether data quality metrics lie within expected bounds. We employ conditional unit tests based on conditional metrics (CMs) that compute data quality signals over specific parts of the ingestion data and therefore allow for a fine-grained detection of errors. A violated conditional unit test specifies a set of erroneous tuples in a natural way: the subrelation that its CM refers to. Unfortunately, the downside of their fine-grained nature is that violating unit tests are often correlated: a single error in an ingestion batch may cause multiple tests (each referring to different parts of the batch) to fail. The key challenge is therefore to untangle this correlation and filter out the most relevant violated conditional unit tests, i.e., tests that identify a core set of erroneous tuples *and* act as an explanation for the errors. We present CM-Explorer, a system that supports data stewards in quickly finding the most relevant violated conditional unit tests. The system consists of three components: (1) a graph explorer for visualizing the correlation structure of the violated unit tests; (2) a relation explorer for browsing the tuples selected by conditional unit tests; and, (3) a history explorer to get insight why conditional unit tests are violated. In this paper, we discuss these components and present the different scenarios that we make available for the demonstration.

PVLDB Reference Format:
Niels Bylois, Frank Neven, and Stijn Vansummeren. CM-Explorer: Dissecting Data Ingestion Problems. PVLDB, 16(12): 3958 - 3961, 2023.
doi:10.14778/3611540.3611595

1 INTRODUCTION

Modern data analytics pipelines continuously collect and ingest new data. Validating the quality of collected data at ingestion time is crucial in such pipelines, for a number of reasons. First, and foremost, the quality of the derived insights, and the decisions driven by them, depend directly on the quality of the collected data [8]. Second, as more and more of the data analysis process is automated, small errors in source data risk propagating to later data consumers (such as machine learning models), which may themselves act as data sources in other processing pipelines—thereby potentially magnifying the error [3]. Finally, data errors may even cause data processing pipelines to crash (e.g., because of null pointer exceptions due to missing data). Machine learning platforms are therefore including explicit data validation components into their pipelines [1, 2, 5].

In recognition of the importance of data quality validation in modern analytics pipelines, several tools have been proposed to aid in automatic validation [3, 7, 8]. Broadly speaking, these tools allow specification, either manually or automatically, of so-called *data unit tests*. When a new batch of data is to be ingested, the registered tests are executed to gauge the batch's quality where failing tests highlight data quality problems. The tests themselves entail computing certain metrics on the data batch (e.g., the minimum or maximum value appearing in a numerical column or the number of distinct elements appearing in a column) and checking that these fall within an expected range.

Unfortunately, these tools suffer from two limitations. First, the data unit tests that they support are based on *global* metrics: metrics that are computed on the entire data batch, or on an entire column in the batch. As the following example shows, they hence only provide *coarse-grained* signals of data quality and are unable to detect *fine-grained* errors, i.e., errors that occur only in a specific (potentially small) part of the batch. Batches with fine-grained errors hence go unnoticed. Second, even when a data unit test signals that a batch has a data quality problem, it does not provide a principled method to identify the part of the batch that is responsible for a test's failure. As such, either the entire batch must be discarded or a human expert must manually identify and subsequently remove or correct the erroneous tuples—which is time-consuming and labor-intensive.

Example 1.1. A public railway company has equipped all of its trains with measurement sensors that record the train's arrival and departure time at each train station. By comparing these times with the time schedule, the train's software computes the corresponding delays (if any). At the end of each day, the measurements of all trains are collected, and ingested in the railway company's data lake. The delays are used to identify hotspot routes, as well as computation of service quality indicators to the government. Train 5437 runs daily from Hasselt (Belgium) to Blankenberge (Belgium). This route is notorious for the delays that it incurs when it passes through the busy Brussels railway stations. As such, train 5437 normally reports non-zero delay. Due to a hardware malfunction on March 15, however, it consistently reports zero delay for this train.

The metrics used by state-of-the-art tools are unable to detect this error. Indeed: because zero delay is *not* an uncommon value when considering the entire ingestion batch (some trains run on

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611595

time), metrics such as min(delay), max(delay), and avg(delay) will not consider zero delay as an anomaly.

It is important to observe that, even if one of the global metrics signals a data quality problem, it is not clear which train or set of trains in the batch cause the problem. For instance, if the unit tests based on min(delay) or max(delay) signal an anomaly then of course we can easily identify the erroneous trains: simply compute the trains whose delay value is below (or above) the expected minimum (resp. maximum) value. However, if a unit test based on avg(delay) signals a problem, then it is unclear how to identify the trains that caused the delay to deviate. □

We have developed an approach for data quality validation that is not only capable of detecting the *fine-grained errors* illustrated above, but also helps in *identifying* responsible erroneous tuples [4]. Our methodology focuses on the setting where a data pipeline regularly ingests batches of external data. As the main technical vehicle underlying our methodology, we introduce *conditional metrics* (CM for short). In contrast to a global metric, a conditional metric only computes its value on a specified subset of tuples in the ingestion batch. A CM m is of the following form

$$m := \mu(Y \mid X = x) \qquad (\dagger)$$

where μ is a metric like MIN, MAX, SUM, MAXDIGITS, ...; Y and X are attributes of the relation under consideration and x is a domain value for X. We refer to (X, x) as the entity in m. The semantics is as follows: m computes the metric μ over the column Y for the subrelation consisting of all tuples where X equals the value x. In the railway example above, for instance, the conditional metric

$$\textsc{avg}(\mathit{Delay} \mid \mathit{Train} = 5437)$$

computes the average delay of train 5437 in the batch, as opposed to the global metric AVG(*Delay*) which computes the average delay of all trains. A conditional unit test then is simply a CM with an associated function that returns true for admissible values and false otherwise. Because of their ability to calculate values for specific entities (e.g., *Train* = 5437), conditional data unit tests may hence detect data quality issues at a finer level of granularity than data unit tests based on global metrics.

Our methodology consists of two phases: (i) a unit test discovery phase and (ii) a monitoring and error identification phase, as schematically illustrated in Figure 1. In the *unit test discovery phase*, we are given a sequence $\overline{R}$ of previously ingested batches and our objective is to automatically derive a set Θ of CM-based data unit tests from $\overline{R}$ such that a yet-to-be-ingested batch B can be considered to be of acceptable quality if it passes all tests in Θ. Specifically, like [7], we assume that the batches in $\overline{R}$ are themselves of acceptable data quality—which is reasonable because they have already been ingested successfully. Under this assumption, we may derive CM-based data unit tests from $\overline{R}$ by considering the set of all possible CMs, and deriving, for each such CM m, a classifier that allows to distinguish expected values of m from anomalies. If m produces an anomaly on a yet-to-be-ingested batch B, then B will be flagged as having a quality issue. To derive the classifier based on the values of m observed in $\overline{R}$, in principle any anomaly detection method [6] may be used. For illustration purposes, Figure 1 mentions some simple anomaly detection methods that produce classifiers that

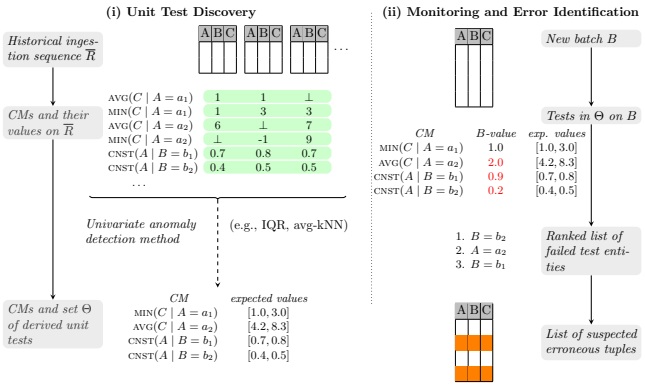


Figure 1: Overview of methodology: (i) *discovery phase* **to derive data unit tests based on conditional metrics; (ii)** *monitoring and error identification phase* **where each new batch is validated and erroneous tuples are identified.**

can be summarized as an expected range of values, where values outside the range are anomalies. Whatever method is chosen, we require that the derived classifier is consistent with our assumption that $\overline{R}$ is of reasonable quality, which implies that the values of m observed in $\overline{R}$ are without significant anomalies.

In the *monitoring and error identification phase*, we take the set Θ of data unit tests derived in the discovery phase, and use this to validate each new ingestion batch B. If all tests in Θ succeed on B then B is deemed to be of acceptable quality. When at least one test in Θ rejects B then our objective is to identify the tuples in B with suspected errors. Every conditional unit test specifies a set of erroneous tuples in a natural way: the subrelation of B that its CM refers to. In our railway example, if a unit test with CM AVG(*Delay* | *Train* = 5437) hence fails, all tuples with *Train* = 5437 could be flagged. Simply flagging *all* the tuples of violated unit tests selects too much, however (i.e., it results in high recall but very low precision). The reason is that violating tests are often correlated: a single error in B may cause multiple tests (each selecting different subrelations) to fail. The key challenge in the monitoring phase, therefore, lies in *ranking* the violated unit tests according to relevance, and from this ranked list of tests *filter* a list of suspected erroneous tuples for further inspection. We have developed multiple metrics to rank and filter violated unit tests in [4].

We illustrate the challenges in analyzing violated unit tests by means of an example. For this, we introduce the bipartite entity-tuple graph as a means to represent the correlation structure of the violated unit tests. This graph consists of two sets of nodes: all entities mentioned in at least one of the violated unit tests and every tuple that is selected by at least one of them. Furthermore, there is an edge from an entity to a tuple when that tuple is selected by the entity. Recall the scenario outlined in Example 1.1, where due to a hardware malfunction train 5437 consistently reports zero delay. Consider the to-be-ingested batch B shown in Figure 2 as well as the failed unit tests ϕ_1, ϕ_2, and ϕ_3 listed there. The entity-tuple graphs is displayed as well. We note that for the sake of this example, we have added an extra attribute *Kind* that lists the kind of service that the train offers. Unit test ϕ_1 fails because the average delay of train

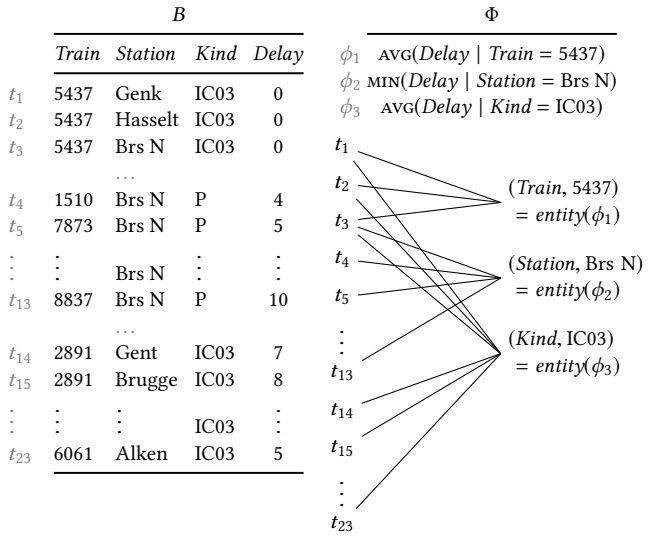


Figure 2: Illustration of a to-be-ingested batch B, failed unit tests Φ, and the corresponding entity-tuple graph.

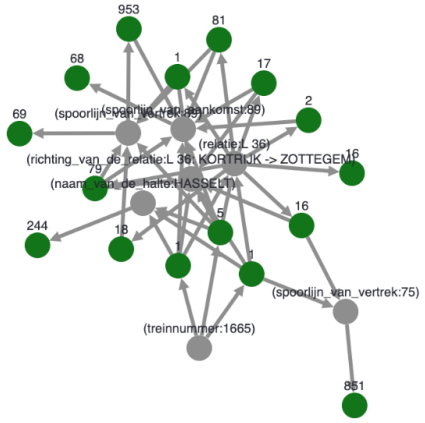


Figure 3: Entity-tuple graph explorer example for a single component for all *flagged* entities.

5437 is now zero, which is unexpected given the historical sequence. However, in this example, the zero delay of train 5437 also causes the minimum delay in station Brussels North (Brs N) to become zero, which is also unexpected, causing ϕ_2 to fail. Similarly, it also causes the average delay of route kind IC03 to become unexpected, failing ϕ_3. The root cause here is the zero delay of train 5437, and ideally we would hence like to filter out ϕ_1 that selects only the tuples t_1–t_3 of train 5437 as suspected erroneous tuples.

We present CM-Explorer, an interactive system that focuses on the monitoring and error identification phase of the methodology described above, and that allows to interactively investigate the observed data quality problems. Using CM-Explorer, a data steward may easily inspect the correlation structure of the violated unit tests via a graph-based representation to identify the most relevant ones for selecting a set of erroneous tuples and serving as an explanation for them. We next describe the components of CM-Explorer that support this (in Section 2), and discuss concrete demonstration scenarios in Section 3.

2 OVERVIEW OF CM-EXPLORER

The CM-Explorer consists of three components: the graph explorer, the CM history explorer, and the relation explorer. The workflow supported by CM-Explorer is as follows. A data steward may first utilise the graph explorer to filter the failed conditional unit tests by selecting tests with high scoring entities and removing tests with entities that refer to column names that are not considered to be important. In a second step, the data steward can zoom into individual entities using the history and relational explorer, to further reduce the conditional unit tests until only those of interest remain.

2.1 Graph explorer

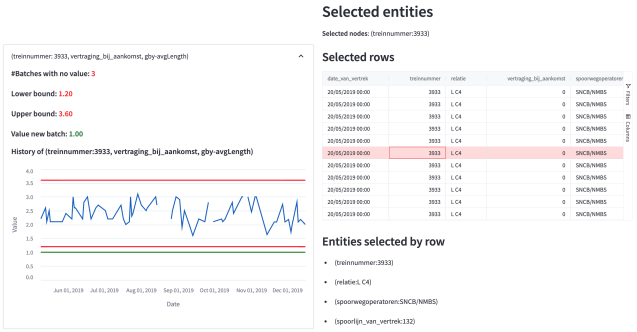
An example of a real-world entity-tuple graph, as inspectable in the graph explorer, is displayed in Figure 3. The graph explorer always shows a contracted version of the entity-tuple graph, where tuple nodes are combined into a single node when they are connected to the same set of entity nodes. Contracted tuple nodes are colored green and carry as label the number of tuples that they represent. Entity nodes are colored grey. The graph explorer supports the data steward in analyzing the entity-tuple graph through a number of filters:

- Filtering on score: As explained above, we developed a number of metrics to rank entities and each of those can be used to filter for high scoring entities (and the associated CMs). When a particular measure is chosen, the score of each entity can be displayed.
- Filtering on entity column name: Consider the conditional metric AVG($Delay$ | $Train$ = 5437). Then $Train$ refers to the entity column name and $Delay$ to the column name the aggregate is applied to. The data steward can filter out specific column names that she regards as uninteresting thereby reducing the size of the entity-tuple graph.

To further support interactive exploration, the graph explorer component allows to remove specific entities from the graph, that, for instance, have been inspected by the components described in the following sections, and for which it has been established that they do not refer to erroneous tuples and can be safely disregarded.

2.2 History explorer

When an entity is selected in the graph explorer, the history view shows all the data points from previous batches in blue, with the lower and upper bounds highlighted in red (see Figure 4a for an example). The value on the current batch (that reported an error) is shown in green. When there is no value for a specific batch in the historical data, this is shown as a gap in the graph. The user can hover over a specific data point to inspect the exact value for that batch. The user can hence explore in more detail why specific



(a) CM history view **(b) Relation explorer**

Figure 4: History view and relation explorer for the delay of train 3933.

conditional metrics have reported an error for that entity and can choose to keep or exclude the corresponding entity from the graph explorer.

2.3 Relation explorer

We refer to Figure 4b for an example of the relation explorer. We next explain the two modes:

From entities to selected tuples: When an entity is selected in the graph explorer, the relation explorer shows the tuples of the batch that are selected by the entity. Together with the history explorer described in Section 2.2, this functionality allows to inspect individual tuples for errors. For example, the history explorer might show that the average length of the delay for train 3933 is too low. The relation explorer can then be used to view the tuples for train 3933 and identify the specific delay that is causing the error.

From tuples to entities: The relation explorer may also be used in the other direction: the user can select a tuple in the relation explorer which will reveal the entities that select the corresponding tuple. This allows to identify which entities are affected by the selected tuple and can be used to correlate the error with other errors that might be reported by other entities.

2.4 Test data generation

For demonstration purposes, we also provide a principled way to generate test data that is similar to what might occur in the real world. This allows the user to insert errors in a new batch of data and run the analysis pipeline to explore the results in CM-Explorer. Modifications are based on the granularity of an entity. The user first picks the entity that should be modified, and specifies the column to change together with the modified value. We also allow partial modification through the specification of an 'edit percentage'. Any number of entities can be modified, and for each entity it is possible to modify any number of columns. An example modification can be specified as follows: set the *delay* value to 0 for 75% of tuples for the entity *train* 3933. This modification will result in the conditional metric *average length of delay* reporting an error for the entity *train* 3933, as shown in Figure 4a.

3 DEMONSTRATION SCENARIO

In this demonstration, participants will be able to interact with our proposed data quality validation approach [4], as embodied in CM-Explorer and described above. We provide the following demonstration scenarios:

Exploration mode. In this scenario the participant is given the opportunity to manually introduce errors in an ingestion batch using the test data generation component described in Section 2.4. The goal of this scenario is to provide a controlled environment in which to explore the different components of CM-Explorer and assess their effectiveness in filtering for the relevant violated conditional unit tests to uncover the introduced errors.

Detective mode. In this scenario, the participant may use CM-Explorer on a number of existing batches with errors. For these batches there is a clear *single* explanation of the error, as for instance is the case for Example 1.1 where due to a hardware malfunction one train consistently reports zero delay. The purpose of this scenario is to showcase the effectiveness of CM-Explorer to uncover errors in ingestion batches.

Inspector Columbo mode. This scenario is similar to detective mode except that a batch now contains *different* types of errors and it is more challenging to find the different errors and their corresponding explanations. The purpose of this scenario is to showcase the effectiveness of CM-Explorer in a more complex setting.

ACKNOWLEDGMENTS

S. Vansummeren was supported by the Bijzonder Onderzoeksfonds (BOF) of Hasselt University under Grant No. BOF20ZAP02. This work is partially funded by the Research Foundation – Flanders (FWO-grant G055219N). The resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation – Flanders (FWO) and the Flemish Government.

REFERENCES

[1] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *SIGKDD (2017)*. 1387–1395.

[2] Joos-Hendrik Boese, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Dustin Lange, David Salinas, Sebastian Schelter, Matthias W. Seeger, and Bernie Wang. 2017. Probabilistic Demand Forecasting at Scale. *Proc. VLDB Endow.* 10, 12 (2017), 1694–1705. https://doi.org/10.14778/3137765.3137775

[3] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *MLSys (2019)*.

[4] Niels Bylois, Frank Neven, and Stijn Vansummeren. 2023. Data Ingestion Validation through Stable Conditional Metrics with Ranking and Filtering. (2023). https://github.com/fneven-uh/cm-paper.

[5] Emily Caveness, Paul Suganthan G. C., Zhuo Peng, Neoklis Polyzotis, Sudip Roy, and Martin Zinkevich. 2020. TensorFlow Data Validation: Data Analysis and Validation in Continuous ML Pipelines. In *SIGMOD (2020)*. ACM, 2793–2796.

[6] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (jul 2009), 58 pages.

[7] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. 2021. Automating Data Quality Validation for Dynamic Data Ingestion. In *EDBT 2021*. OpenProceedings.org, 61–72.

[8] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *Proc. VLDB Endow.* 11, 12 (2018), 1781–1794.