Debugging Large-Scale Data Science Pipelines using Dagger

El Kindi Rezig⋆ Ashrita Brahmaroutu⋄ Nesime Tatbul⋆⋄ Mourad Ouzzani♣ Nan Tang♣
Timothy Mattson⋄ Samuel Madden⋆ Michael Stonebraker⋆

⋆MIT CSAIL ⋄Intel ♣Qatar Computing Research Institute, HBKU
{elkindi, tatbul, madden, stonebraker}@csail.mit.edu
{ashrita.brahmaroutu, timothy.g.mattson}@intel.com {mouzzani, ntang}@hbku.edu.qa

ABSTRACT

Data pipelines are the new code. Consequently, data scientists need new tools to support the often time-consuming process of debugging their pipelines. We introduce *Dagger*, an end-to-end system to debug and mitigate data-centric errors in data pipelines, such as a data transformation gone wrong or a classifier underperforming due to noisy training data. *Dagger* supports inter-module debugging, where the pipeline blocks are treated as black boxes, as well as intra-module debugging, where users can debug data objects in Python scripts (e.g., DataFrames). In this demo, we will walk the audience through a rich, real-world business intelligence use case from our industrial collaborators at Intel, to highlight how *Dagger* enables data scientists to productively identify and mitigate data-centric problems at different stages of pipeline development.

1. INTRODUCTION

Data scientists use data pipelines for a myriad of applications (e.g., business forecasting, medical diagnosis). Python libraries such as *Pandas*, *scikit-learn* have matured significantly over the past few years. As a result, authoring data science pipelines has become more accessible than ever.

Data scientists typically have a collection of connected Python modules to handle different tasks in a data pipeline (e.g., data preparation, data analytics, etc.) [6, 8, 11]. Oftentimes, pipeline errors stem from the data (input or intermediate data), and not the code [5, 3, 8, 10]. Debugging pipelines at the data level is still in its infancy. There is a lack of tools to assist in debugging data handled in the pipeline. For instance, when data produced by a particular pipeline module is erroneous (e.g., bad data transformation or missing values), then the rest of the pipeline will likely produce bad data as well.

Pipeline data exists at two levels: (1) inter-module: data that is passed across the pipeline modules (e.g., files) through their I/O interfaces; and (2) intra-module: Python data structures (e.g., Pandas DataFrames) that are used inside the module. Data errors can occur at either one of these two levels.

Proceedings of the VLDB Endowment, Vol. , No.
ISSN 2150-8097.
DOI:

While very mature, code debuggers are not ideal to find problems that stem from the data. To address this shortcoming, we have built *Dagger* [7], an end-to-end system that supports data debugging at two granularities: (1) inter-module debugging, where users can treat the pipeline modules as black boxes and only debug the data in the input/output of those modules; and (2) intra-module debugging, where users debug code-handled data. We focus on scripts written in Python for the latter.

Dagger offers data scientists a set of primitives to query and debug data pipelines. For instance, if the training data has records with missing values in the attribute `salary`, a data scientist might ask the question: "What if I train two models, one with data that includes all the rows, and another with data that excludes the rows without a `salary` value?" (using Dagger's *split* primitive), and then compare the models in terms of accuracy (using Dagger's *compare* primitive). *Dagger* makes it easy to express such ad-hoc queries allowing data scientists to experiment with various what-if questions.

We propose to demonstrate how *Dagger* helps identify and debug a real-world data science pipeline. The audience will experience firsthand how *Dagger* significantly reduces the number of iterations needed to converge to a satisfactory pipeline by reducing the time needed to identify data-centric problems. Specifically, we will use a live pipeline debugging session of a business forecasting scenario and show how *Dagger* addresses various data-centric problems that typically happen during development (e.g., suspect training data, erroneous values). In the spirit of *GDB* [1], *Dagger* is a command-line tool where users issue statements to perform various debugging operations. For the purpose of the demo, we will show a graphical user interface to author the pipelines and to run *Dagger* statements on them.

There is currently a plethora of tools to address different facets of pipeline development. For instance, a popular choice to author and debug data pipelines is through Jupyter notebooks [2]. Such notebooks allow the mixing of scripts (cells), descriptive text, and code output in one document. However, notebooks are not specifically geared towards data debugging, i.e., users still need to write code to test for different data-centric scenarios. Another line of work focuses on tracing and exploring data provenance to expose potential data-centric problems. VisTrails tracks data provenance and allows users to interact with it through visual interfaces [9]. Vizier allows using multiple interfaces (e.g., spreadsheets and Jupyter notebooks) to interact with pipeline data [4]. While those tools are useful to explore pipeline data, they do not provide explicit data debugging primitives, i.e., the burden is on the user to take note of errors and anomalies in the data. *Dagger* is a novel tool that takes away this burden from the users, thereby improving their productivity.

2. SYSTEM OVERVIEW

Figure 1 illustrates the architecture of *Dagger*. *Dagger* takes as input the Python modules of a given pipeline, called pipeline blocks, and then runs these blocks and logs their data objects at run time. We currently log tabular data from *Pandas DataFrames* and *numpy* arrays. The logged data is stored in a Postgres database. Users then use a SQL-like language to declaratively query, inspect, and debug the data.

Workflow manager. Users can load their pipelines (i.e., Python scripts) in *Dagger* with no additional coding effort. Users just need to tag Python code blocks (i.e., a range of line numbers) in their scripts (or use the whole script files) where they want data to be tracked and stored by *Dagger* for later querying. For example, the following Dagger statements create pipeline blocks (named *b*1 and *DEDUP*) from two code blocks (in *preprocessing.py* and *dedup.py*):

```
CREATE BLOCK b1 FOR PIPELINE P1: preprocessing.py:1-600
CREATE BLOCK DEDPUP FOR PIPELINE P1: dedup.py:10-144
```

Logging manager. *Dagger* logs data from the tagged blocks into a Postgres database. The logging manager tracks and stores values of tabular data (e.g., Pandas DataFrames) across the pipeline blocks. It also stores inter-module data. Additionally, because the data generated at each run can be overwhelmingly large, *Dagger* strives to avoid storing duplicate data by dividing data objects into a "base" and "delta". The former is a data subset that is repeated in two or more data objects, and the latter is the "new" data that is added to the base. This way, we can reconstruct a data object by combining its base and delta. We refer to this data compression scheme as "delta logging".

Debugging primitives. *Dagger* exposes four primitives to debug pipeline data:

- *data breakpoints* are used to test data assertions across the pipeline blocks (e.g., salary should not exceed a certain value).
- *split* is used at runtime to divide an input table into multiple partitions that are used as input to the subsequent pipeline blocks.
- *data generalization* is for gathering data points that are similar to sample data points (e.g., find values that exhibit the same errors, or find more "good" values to use as input).
- *compare* is designed to compare two pipelines by the data they generate. This helps in identifying pipelines that generate the same data even though they might be different (e.g., different parameters or modules).

In Section 3, we demonstrate how those primitives are used to debug various data-centric problems in a real-world pipeline.

Interaction language. *Dagger* features a SQL-like language to express debugging queries and statements. We refer to this language as DQL (Dagger Query Language). The following statement generates two runs of the block `DEDUP` in pipeline `P1` with dataset `salary_data`: One run with records whose `salary` is less than 60,000, and the other run with records whose `salary` is greater than or equal to 60,000.

```
    RUN BLOCK DEDUP IN P1 WITH SPLIT salary_data
    WHERE salary < 60,000
```

We now show how *Dagger* helps accelerate pipeline development in a real-world use case.

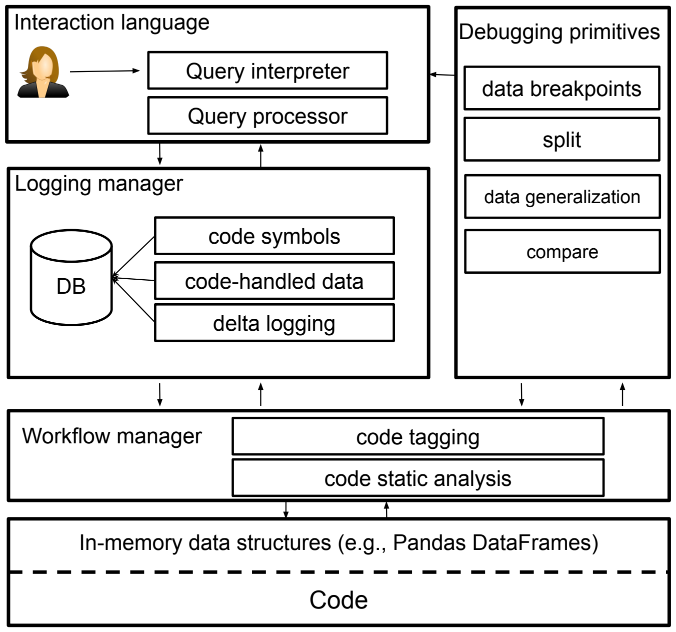


Figure 1: Dagger Architecture

3. DEMONSTRATION SCENARIO

Our demo will focus on a machine learning pipeline (Figure 2) that predicts which marketing campaigns are best for which customers. This prediction is based on historical data from the outcome of previous marketing campaigns. The goal is to establish models to forecast where marketing budget should be allocated and how we can optimize existing marketing campaigns. We want to assess whether certain marketing campaigns are bringing in more return for certain customers (e.g., sales impact) as well as predicting what this trend will look like over time.

3.1 Dataset

Our core data source is the `Customers` table with the following schema elements:

Spend Type	Customer ID	Customer Name	Year
Month	Activity Category	Vertical Industry	Product Type
Geo	Initiative/Campaign	Marketing Program	Payment
Quarter			

The table contains mostly categorical variables and subcategories of other existing variables. These include the marketing program and campaign, the type of spending, the target product, vertical industry along with the investment made, the geographical region in which the customer is located, the customer name, time dependencies - yearly, quarterly, monthly - and the amount invested (numerical), among others. We use these raw attributes to construct, aggregate, and transform the data before feeding it to a classifier. This table contains well over a million rows, dating from 2010 to the present.

3.2 Pipeline Blocks

Figure 2 illustrates the key pipeline blocks for our use case.

Data preparation: This includes imputing missing values, finding and correcting incomplete fields, and standardizing value representations (e.g., MA instead of Mass. or Massachusetts). Additionally, some numerical fields may be aggregated over various time granularities (months, quarters, etc.). From this step, the data scientist selects a subset of the data to label. We refer to those labels as seed labels.

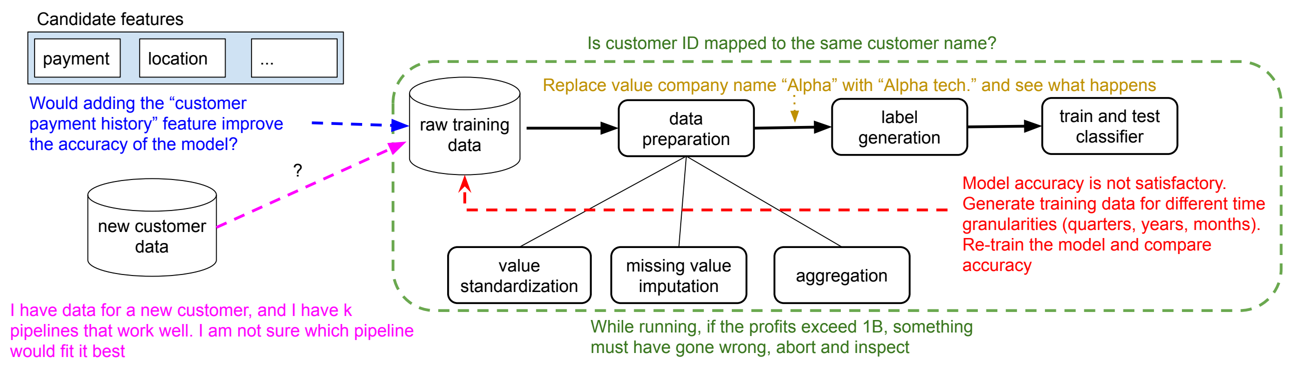

Candidate features
payment
location
...
Would adding the "customer payment history" feature improve the accuracy of the model?
?
new customer data
I have data for a new customer, and I have k pipelines that work well. I am not sure which pipeline would fit it best
raw training data
Is customer ID mapped to the same customer name?
Replace value company name "Alpha" with "Alpha tech." and see what happens
data preparation
label generation
train and test classifier
value standardization
missing value imputation
aggregation
Model accuracy is not satisfactory. Generate training data for different time granularities (quarters, years, months). Re-train the model and compare accuracy
While running, if the profits exceed 1B, something must have gone wrong, abort and inspect

Figure 2: A typical use case pipeline from a data scientist (referred to in the text as "Lou")

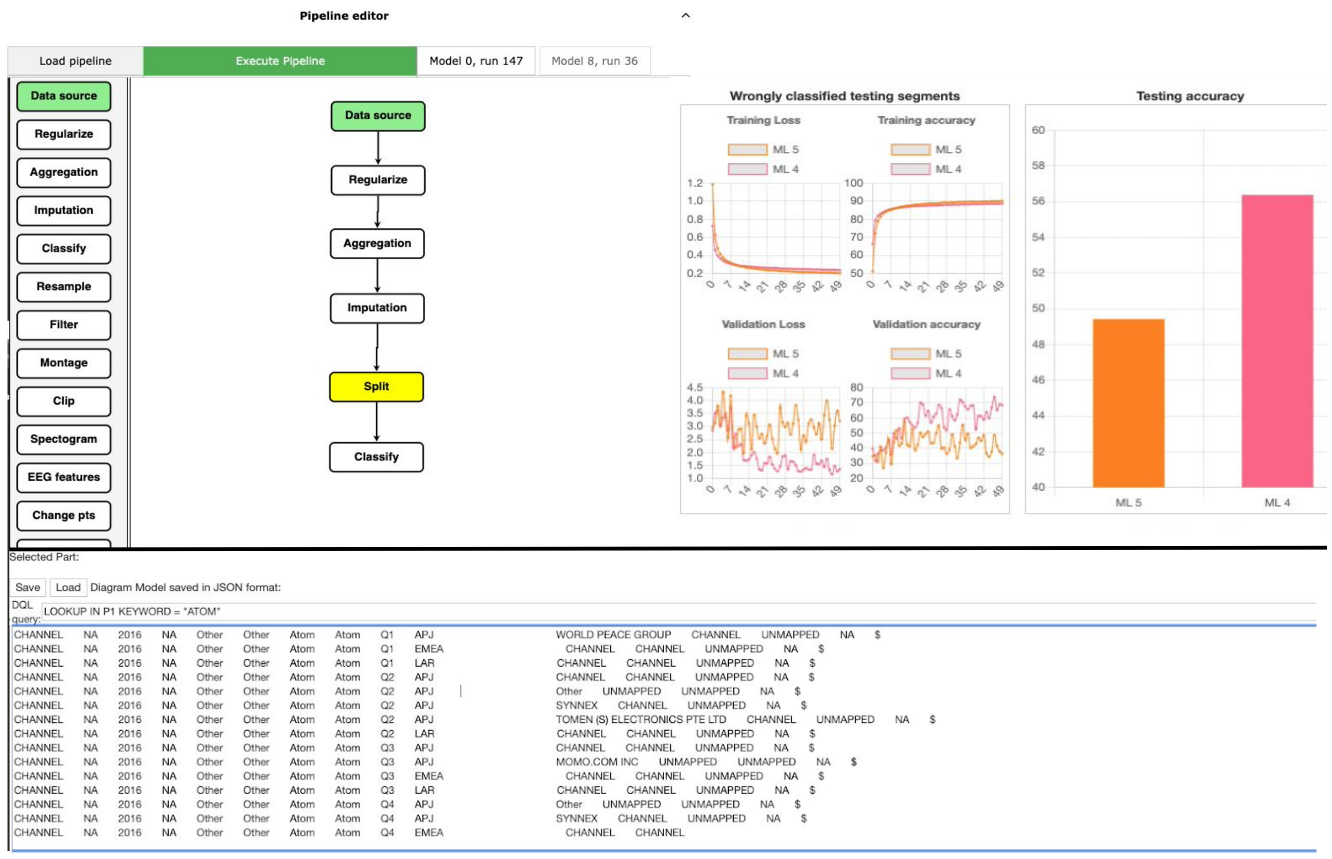
Label creation and ranking: We use a similarity model with various features, such as customer type, vertical industry, and validity of the marketing campaign to generate more training data by including data points that are similar to those in the seed labels.
Modeling and output: We train multiple models at different levels of aggregation (forecasting for short term vs. long term) using logistic regression and Recurrent Neural Networks (RNNs), and assess their performance. We can use these models to estimate how marketing programs will fare in the future (based on similar programs).

3.3 Debugging Scenario

A data scientist, Lou, prepares the data pipeline in Figure 2 and starts noticing a few problems concerning the data produced by different pipeline blocks. We describe some of these problems and how *Dagger* helps Lou address them with minimal effort.
Maintaining data invariants throughout the pipeline: During each stage of the data pipeline, data objects (intra-module and inter-module) go through multiple transformations and iterations, sometimes resulting in data errors due to several factors including bad input data, code bugs, and bad module parameters. However, Lou knows of a few assertions that must always hold on the data and can express them using the *data breakpoint* primitive in *Dagger*. We provide a couple of examples of breakpoints below.
Customer ID and name mapping: Rows with the same customer ID must carry the same customer name value. There may be points in the pipeline where that 1:1 mapping may be violated; due to typos, or accidental changes in values. In this scenario, customer name may take several forms in the data. For example, a company named "Alpha" may be denoted as "Alpha Technologies" or "Alpha Inc." in other rows or sections of the data. For this reason, we may refer to the customer ID to identify a particular company. In *DQL*, the breakpoint can be created with the following statement (for any pair of records `t1` and `t2` in pipeline `P1`, if `t1` and `t2` share the same company ID, they must share the same company name):

```
CREATE BREAKPOINT B1 ON PIPELINE P1 WHERE
P1.t1.company_id = P1.t2.company_id AND
P1.t1.company_name = P1.t2.company_name
```

Profit and loss assertion: Since the data includes various marketing programs, we want to make sure certain numerical values do not exceed certain thresholds. We first construct a simple profit and loss matrix (how much is earned vs. how much is lost per program) to understand the return on investment on campaigns. Processing each campaign takes time (data preparation and training), so we would like to skip campaigns that show low returns. Data breakpoints are useful in this case, as they allow for identification of failed marketing campaigns (i.e., loss $>>$ profit) and abnormal output (e.g., profit $>$ 1B over the past 3 months).
Diagnosing what is wrong with the training data: Lou notices that the trained model is producing poor results (i.e., low testing accuracy). A problem with the training data is suspected. Lou can use the *split* primitive in *Dagger* to test this hypothesis.
Experimenting with different time granularities: Lou knows that in many business intelligence prediction applications, the sampling frequency of the data impacts the model output. Typically, more granular data, in this case, weekly and monthly samples, will produce a shorter term forecast to use. In other cases, using quarterly or yearly data aggregations establishes a longer term forecast. The variation in forecasts may result in differences in executive decisions, which has direct impact on the business. This leads to uncertainty within the data itself, and the forecasting model. Splitting the data at different time granularities allows Lou to test variability in model outputs and assess this uncertainty in operational outcomes. To do this, Lou aggregates by summing the total investment along each corporate partner and marketing program at a weekly level. Lou then increases the time granularity and does the same for monthly, quarterly, and yearly data. Doing this manually can be time consuming, as we separate and run different pipelines and algorithms on each of the time frequencies. *Dagger* makes this easy by splitting the training data on those different granularities and automatically invoking the training block on each one of them.
Taking advantage of what has already been run: Lou goes through many iterations building and refining the data pipeline. If Lou makes a change in a previously run pipeline (e.g., new modules parameters), it is important to know whether this new pipeline would produce different results from a pipeline that was already run. This is important because running those pipelines may take a long time. *Dagger* provides a primitive, *compare* to compare data intermediates between two pipelines. Lou can see, while the pipeline is running, whether the data produced so far is exactly the same as the one produced in a previous run. Lou can then, abort the execution.
Is this new feature useful? Lou would like to see if the addition of a new feature would have any impact. For example, Lou anticipated that using the activity status (invalid or valid payments) could have some impact on the accuracy of the model. Lou adds this feature to the data, but notices right after the data preparation step that the "profits and losses" are the exact same as the ones produced on data without this feature. So, Lou aborts the execution before even training a model, since the model would most likely be the same as the one obtained from a previous run.



```
LOOKUP IN P1 KEYWORD = "ATOM"
```

```
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q1  APJ    WORLD PEACE GROUP  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q1  EMEA     CHANNEL  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q1  LAR    CHANNEL  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q2  APJ    CHANNEL  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q2  APJ    Other  UNMAPPED  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q2  APJ    SYNNEX  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q2  APJ    TOMEN (S) ELECTRONICS PTE LTD  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q2  LAR    CHANNEL  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q3  APJ    CHANNEL  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q3  APJ    MOMO.COM INC  UNMAPPED  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q3  EMEA     CHANNEL  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q3  LAR    CHANNEL  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q4  APJ    Other  UNMAPPED  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q4  APJ    SYNNEX  CHANNEL  UNMAPPED  NA  $
CHANNEL  NA  2016  NA  Other  Other  Atom  Atom  Q4  EMEA     CHANNEL  CHANNEL
```

Figure 3: Dagger User Interface. Upper-left: Pipeline editor. Upper-right: ML performance charts. Bottom: Dagger console

Finding similar data points: Lou often notices data points at different stages of the pipeline that are worth inspecting, e.g., outliers and errors. To inspect those points further, Lou first needs to find other data points that are similar to them. Additionally, given a new batch of records for a new customer, Lou would need to decide which pipeline would fit this data best (assuming Lou has a list of K pipelines dealing with different customers' data).

Which pipeline is best for new data? Lou already has a list of K pipelines dealing with different classes of customers (e.g., a pipeline handling customer data pulled from a legacy HR system requires different modules than one handling customer data with a current HR system). When a new batch of records for a new customer comes in, Lou needs to quickly tell which of the K pipelines has input data that is most similar to the incoming data. The similarity functions are defined by Lou for the different attributes of the tables.

3.4 User Interface

Figure 3 illustrates the user interface of *Dagger*. The upper-left side of the interface is dedicated to editing pipelines, while the upper-right side contains charts to show the audience how well different ML models perform. The bottom pane contains the console from which users can write *DQL* statements (e.g., `LOOKUP` performs a keyword lookup) and examine their output. Demo attendees will have an opportunity to interact with this interface and play the role of "Lou".

4. REFERENCES

[1] GNU Debugger. `https://www.gnu.org/software/gdb/`. Accessed: March 2020.

[2] Jupyter Notebooks. `https://jupyter.org/`. Accessed: March 2020.

[3] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting Data Errors: Where are we and what needs to be done? *PVLDB*, 2016.

[4] M. Brachmann, C. Bautista, S. Castelo, S. Feng, J. Freire, B. Glavic, O. Kennedy, H. Müeller, R. Rampin, W. Spoth, and Y. Yang. Data Debugging and Exploration with Vizier. In *SIGMOD*, 2019.

[5] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The Data Civilizer System. In *CIDR*, 2017.

[6] T. Kraska. Northstar: An Interactive Data Science System. *PVLDB*, 2018.

[7] E. K. Rezig, L. Cao, G. Simonini, M. Schoemans, S. Madden, N. Tang, M. Ouzzani, and M. Stonebraker. Dagger: A Data (not code) Debugger. In *CIDR*, 2020.

[8] E. K. Rezig, L. Cao, M. Stonebraker, G. Simonini, W. Tao, S. Madden, M. Ouzzani, N. Tang, and A. K. Elmagarmid. Data Civilizer 2.0: A Holistic Framework for Data Preparation and Analytics. In *PVLDB*, 2019.

[9] C. T. Silva, J. Freire, E. Santos, and E. W. Anderson. Provenance-Enabled Data Exploration and Visualization with VisTrails. In *SIBGRAPI Conference on Graphics, Patterns and Images*, pages 1–9, 2010.

[10] M. Vartak, J. M. F. da Trindade, S. Madden, and M. Zaharia. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *SIGMOD*, pages 1285–1300, 2018.

[11] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. G. Parameswaran. Helix: Accelerating Human-in-the-loop Machine Learning. *PVLDB*, 2018.